IMPLEMENTATION OF A CONSTRAINT MACHINE FOR PRISM - A PARALLEL PROBLEM SOLVER

by
William P. Bradley

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
1984

UNCLASS
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFIT/CI/NR 84-81T	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): Implementation Of A Constraint Machine For Prism - A Parallel Problem Solver	5. TYPE OF REPORT & PERIOD COVERED: THESIS/DISSERTATION
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): William P. Bradley	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: AFIT STUDENT AT: University of Maryland	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: AFIT/NR, WPAFB OH 45433	12. REPORT DATE: 1984
	13. NUMBER OF PAGES: 141
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASS
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DTIC ELECTE NOV 26 1984 S D B

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1

6 Nov 84

LYNN E. WOLAVER
Dean for Research and
Professional Development
AFIT, Wright-Patterson AFB OH

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASS
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT

Title of Thesis: IMPLEMENTATION OF A CONSTRAINT MACHINE FOR PRISM - A PARALLEL PROBLEM SOLVER

William P. Bradley, Master of Science, 1984

Thesis directed by: Jack Minker
Professor
Department of Computer Science

This thesis describes the implementation of a constraint machine for a parallel inference system, PRISM. PRISM is a logic programming system designed to run on a new parallel computer called ZMOB. PRISM consists of three different kinds of machine, problem solving, intensional database and extensional database machines. Each will run on separate processors on the ZMOB, and there will be many of each type, all cooperating to solve a problem.

The constraint machine is a fourth class of machine for PRISM. It will use a database of integrity constraints, supplied by the user, to help the problem solvers. It will do so by pruning branches from the problem solvers' goal tree whenever a goal node violates an integrity constraint. This will prevent the problem solver from exploring some paths that will ultimately fail. The constraint machine uses a subsumption algorithm to determine if a goal node violates an integrity constraint. It is hoped that the constraint machine will improve the performance of the PRISM system.

Author: William P. Bradley, Major, USAF, BSC

Pages: 141
Degree: Master of Science
School: University of Maryland
Year: 1984

AFIT/CI/NR 84-81T

AFIT RESEARCH ASSESSMENT

The purpose of this questionnaire is to ascertain the value and/or contribution of research accomplished by students or faculty of the Air Force Institute of Technology (AU). It would be greatly appreciated if you would complete the following questionnaire and return it to:

AFIT/NR
Wright-Patterson AFB OH 45433

RESEARCH TITLE: Implementation Of A Constraint Machine For Prism - A Parallel Problem Solver

AUTHOR: William P. Bradley

RESEARCH ASSESSMENT QUESTIONS:

1. Did this research contribute to a current Air Force project?

( ) a. YES ( ) b. NO

2. Do you believe this research topic is significant enough that it would have been researched (or contracted) by your organization or another agency if AFIT had not?

( ) a. YES ( ) b. NO

3. The benefits of AFIT research can often be expressed by the equivalent value that your agency achieved/received by virtue of AFIT performing the research. Can you estimate what this research would have cost if it had been accomplished under contract or if it had been done in-house in terms of manpower and/or dollars?

( ) a. MAN-YEARS _____ ( ) b. $_____

4. Often it is not possible to attach equivalent dollar values to research, although the results of the research may, in fact, be important. Whether or not you were able to establish an equivalent value for this research (3. above), what is your estimate of its significance?

( ) a. HIGHLY SIGNIFICANT ( ) b. SIGNIFICANT ( ) c. SLIGHTLY SIGNIFICANT ( ) d. OF NO SIGNIFICANCE

5. AFIT welcomes any further comments you may have on the above questions, or any additional details concerning the current application, future potential, or other value of this research. Please use the bottom part of this questionnaire for your statement(s).

NAME GRADE POSITION

ORGANIZATION LOCATION

STATEMENT(s):

FOLD DOWN ON OUTSIDE - SEAL WITH TAPE

AFIT/NR
WRIGHT-PATTERSON AFB OH 45433

OFFICIAL BUSINESS
PENALTY FOR PRIVATE USE, $300

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 73236 WASHINGTON D.C.

POSTAGE WILL BE PAID BY ADDRESSEE

AFIT/ DAA
Wright-Patterson AFB OH 45433

FOLD IN

THE GRADUATE SCHOOL

University of Maryland
College Park

Report of Examining Committee

Date: August 10, 1984

Name of candidate William P. Bradley

Degree sought M.S. Date of Oral Examination (X) August 18, 1984

Examination and dissertation approved by the following committee:

Chairman: Jack Minker

Donald Perlis

Representative of the Graduate Dean:

Wm F Atchison

(To be returned to Graduate School Office as soon as signed by the Committee)

APPROVAL SHEET

Title of Thesis: IMPLEMENTATION OF A CONSTRAINT MACHINE FOR PRISM - A PARALLEL PROBLEM SOLVER

Name of Candidate: William P. Bradley
Master of Science, 1984

Thesis and Abstract Approved: Jack Minker

Jack Minker
Professor
Department of Computer Science

Date Approved: 8/17/84

CURRICULUM VITAE

Name: William Pierce Bradley.

PII Redacted

[redacted]

Degree and date to be conferred: M.S., 1984.

[redacted]

[redacted]

Secondary education: Bethesda-Chevy Chase High School
Bethesda, Maryland 1965.

Collegiate institutions attended	Dates	Degree	Date of Degree
Emory University	1965-69	B.S.	1969
Emory University	1969-71	M.S.	1971
University of Maryland	1982-84	M.S.	1984

Major: Computer Science.

Professional positions held: Major, United States Air Force
Lackland AFB, Texas 78236.

ABSTRACT

Title of Thesis: IMPLEMENTATION OF A CONSTRAINT MACHINE FOR PRISM - A PARALLEL PROBLEM SOLVER

William P. Bradley, Master of Science, 1984

Thesis directed by: Jack Minker
Professor
Department of Computer Science

This thesis describes the implementation of a constraint machine for a parallel inference system, PRISM. PRISM is a logic programming system designed to run on a new parallel computer called ZMOB. PRISM consists of three different kinds of machine, problem solving, intensional database and extensional database machines. Each will run on separate processors on the ZMOB, and there will be many of each type, all cooperating to solve a problem.

The constraint machine is a fourth class of machine for PRISM. It will use a database of integrity constraints, supplied by the user, to help the problem solvers. It will do so by pruning branches from the problem solvers' goal tree whenever a goal node violates an integrity constraint. This will prevent the problem solver from exploring some paths that will ultimately fail. The constraint machine uses a subsumption algorithm to determine if a goal node violates an integrity constraint. It is hoped that the constraint machine will improve the performance of the PRISM system.

Dedicated to

my wife Deborah

for her patience, love and understanding during the

preparation of this thesis.

ACKNOWLEDGEMENTS

I wish to express my thanks to Dr. Jack Minker for his guidance and assistance during my graduate school years. Also, I owe thanks to Deepak Sherlekar, Sharma Chakravarthy, Madhur Kohli, Simon Kasif, Rich Piazza and all of the PRISM group for the PRISM library and their support. Finally, I am indebted to the United States Air Force for supporting me during my graduate studies.

TABLE OF CONTENTS

1. INTRODUCTION.

Logic and logic programming have generated considerable interest in the computer science and AI communities. Partly this is due to the ability to express facts and relationship about objects in the real world in a natural way using logic. Also, there is a need to begin exploiting parallel computer architectures. This makes logic programming look attractive. Logic programs do not specify the sequence of execution and so are readily adaptable to parallel architectures [Pereira 1978], [van Emden 1976].

A major cause of inefficiency in executing logic programs is that the search space is large and often contains paths that ultimately result in failure. Typical logic programming systems such as PROLOG [Roussel 1975] [Roberts 1977] use backtracking to find another path to explore upon failure. Kohli et al [Kohli 1983] presented a theory of using integrity constraints to guide the execution of function free logic programs. This theory can be applied to conventional or parallel execution of logic programs. Until now, this theory has remained largely untested or implemented in logic programming systems.

The Parallel Inference System, PRISM is under development by Minker et al [Eisinger 1981] [Minker 1982] [Kasif 1983] at the University of Maryland. This system is to be implemented on a new parallel machine, called ZMOB [Rieger

1980,1981a,b]. In the PRISM system, there are several different types of machines that run on separate processors and cooperate to execute a logic program. These processors are Problem Solvers (PSM), Intensional Database (IDB) and Extensional Database (EDB) machines. Currently, the system is implemented on a VAX 11/780 using a ZMOB belt simulator. However, no use of integrity constraints has been made on this system.

Futo [Futo 1984], based upon Kohli's earlier work, developed the concept of a Constraint Machine (CM). This machine is another module that would run on a separate processor in the PRISM system and use integrity constraints to guide the PSMs in their search for solutions. The purpose of the CM is to help a PSM prune branches from its goal tree based upon a database of integrity constraints.

1.1. Contributions of this Thesis.

This thesis describes an implementation if a Constraint Machine and an associated constraint machine compiler. The CM has been coded in C and runs on the VAX 11/780. It has been designed to run in the PRISM simulated system on the VAX. It presents the opportunity to test the theory that integrity constraints can be used to improve the search of a logic program interpreter. The compiler is an integral part of the CM since it is necessary to compile the user supplied database of integrity constraints for use by the constraint

machine during execution.

1.2 Outline of the Thesis.

Section 2 is a brief introduction into logic programming and in particular subsumption; the ZMOB on which the PRISM system is to ultimately run; and the current PRISM system itself. Section 3 describes the constraint machine compiler and database of integrity constraints. Section 4 describes the constraint machine, its data structures and communications with the rest of the PRISM system. Section 5 describes future needs of the PRISM system and the CM in order to make the CM an integral part of the PRISM system. Section 6 is a summary of the thesis. Finally, there are 4 appendices. Appendix A contains the grammar for the constraint database. Appendix B is the constraint machine compiler code. Appendix C is the code of the CM itself. Both Appendices B and C are C code source listings.

2. BACKGROUND - LOGIC, ZMOB, PRISM.

Logic is thought by many to be the parallel programming language of the future. ZMOB is a new parallel computer architecture. PRISM is a system designed to take advantage of parallel logic programming and the ZMOB parallel architecture.

2.1. Introduction to Logic and Logic Programming.

In this section, I give a brief introduction to the clausal form of logic and logic programming. For further details the reader should see Kowalski [Kowalski 1979] or Chang and Lee [Chang 1973] for logic and Clocksin [Clocksin 1981] for logic programming.

2.1.1. Clausal Form of Logic.

The first order predicate calculus has always been a good representation for knowledge in the real world. Unfortunately, unrestricted logic is difficult to deal with in a computer program because of the variety of ways of representing the same facts. Usually, all logic statements are first converted into clausal form. An algorithm to do this is presented in any introductory logic text. A clause is defined as a disjunction of literals. In turn, a literal is defined as an atom or negation of an atom. An atom is a predicate symbol and its associated terms. Terms are variables, constants or functions of terms. Thus, some clauses

are:

1. Father(x) || Mother(x)
2. P(f(y)) || R("john")
3. ~Parent("john",y)
4. ~P(x) || Q(y) || R(z) || ~T(w)

where ~ stands for NOT and || stands for logical OR. Typically predicates begin with capital letters, variables are lower case letters near the end of the alphabet, functions are lower case letters in the range of f - h and constants are strings.

Clauses can be rewritten using implication and the identity ~P || Q = P -> Q, so that 4 above becomes:

P(x) & T(w) -> Q(y) || R(z)

Where & stands for logical AND and -> stands for "implies". This is read as "if P of x and T of w are true then so is Q of y and R of z. Taking the notation used in logic programming, particularly PROLOG this would be rewritten as:

Q(y) || R(z) <- P(x),T(w)

where a comma has been substituted for the logical AND. This is then read as "Q of y or R of z is true if P of x and T of w are true." If we restrict our clauses to having only one positive literal, then they are called Horn clauses.

Horn clauses have the standard form:

H(x1,x2,...,xn) <- P1(x1,x2,...xn)...Pm(x1,x2,...xk)

This says that H is true if all of the Pi are true. The literal on the left of the <- is called the head and the literals on the right comprise the body of the clause. Most logic programming is restricted to Horn clauses. Obviously, there are 4 types of Horn clauses depending on whether there are literals on one or both sides of the <-. A Horn clause with no body such as:

Father("chip","karen") <-

states a fact or assertion that "chip" is the father of "karen". A Horn clause with no head represents negated information and is called a goal clause or constraint.

<- Father("chip","sam")

The above clause states that it is not the case that "chip" is the father of "sam".

A Horn clause with both a body and head is called an axiom or procedure.

Parent(x,y) <- Father(x,y)

Parent(x,y) <- Mother(x,y)

These procedures state that x is the parent of y if x is the father of y or x is the parent of y if x is the mother of y

respectively. Notice that these two procedures can be used to determine if x is the parent of y and either can be tried first or perhaps both in parallel. The first succeeding would be the desired answer.

Finally, the Horn clause with neither a head nor body:

<-

represents the null clause and is always false.

2.1.2 Resolution.

Just having the clause (or Horn clause) form of representing data alone is not very useful. Resolution, developed by J. Alan Robinson, was a major contribution to the use of logic in computer science [Robinson 1965]. Resolution is an inference mechanism that can be used with clauses to deduce new information. Basically, resolution says that if a set of clauses is satisfiable, then any clause formed by combining two clauses and eliminating a pair of matching but complimentary literals is also true. For example, using propositional formulas, and given:

(P || Q || R) , (~P || T)

we can derive:

(Q || R || T)

This is done by forming the disjunction of all literals in

both clauses and eliminating the complementary pair, P and ~P. If a set of clauses is inconsistent, then the null clause represented by [] or <-, can be derived from the set by resolution.

In actual use in logic programming, resolution refutation is used. Suppose we are given a set of consistent clauses and a goal clause. We wish to determine if the goal clause is a consequent of the initial set. The procedure used is to negate the goal and add it to the set of clauses. If the goal is a theorem of the given set, adding the negated theorem will make the set inconsistent. Hence the null clause can be derived by resolution. Taking an example with propositional formula, suppose the given set is:

	PROLOG form	clause form
1.	P <- Q	P || ~Q
2.	Q <-	Q

and we want to know if P is true. To do this by resolution refutation, negate P to <- P or ~P and add it to the set of clauses.

3.	<- P	~P

Resolving 1 and 3 above in each case gives:

4.	<- Q	~Q

Finally resolving 2 with 4 gives the null clause <- or []. Thus we have shown that by adding the negated goal to the set, we have made the set inconsistent. Therefore, the unnegated goal is a theorem of the original set.

2.1.3. Unification.

In propositional logic, resolution is fairly straight forward. However, in predicate calculus, there is the problem of matching terms in the literal as well as the predicate symbol. The process which accomplishes this is called unification. The unification principle says that two literals are unifiable if there is a substitution of terms for variables that makes the two literals identical. Thus given two literals:

P(x,g(y),"a") and P(z,g("b"),w)

they are unifiable if we substitute "b" for y, "a" for w, and x for z. This set of substitutions is called a unifier or substitution set and usually written as:

{x/z, "b"/y, "a"/w}

To solve a problem in logic programming using unification and resolution would proceed as follows. Start with the goal or negated theorem. Pick a literal in it. Find a procedure or assertion with the same head predicate symbol. Determine if the two literals are unifiable. If so, apply the substitution to both clauses. Expand the goal by the

body of the procedure. Repeat this until all literals in the goal are solved or cannot be solved. If all are solvable, then the goal is true. By a process of called answer extraction, the values of any substituted variables can be obtained. In resolution, there are usually many ways to generate the resolvent clauses. Many will not lead to a solution and it is unlikely that the correct solution will be found the first time. Most PROLOG systems employ backtracking to erase substitutions that ultimately were not provable.

2.1.4. Subsumption.

By definition, a clause C is said to subsume another clause D, if there is a substitution of C such that it becomes a subset of the clause D. For example, the following clauses on the left subsume the clause on the right.

subsuming clause	subsumed clause	substitution
P(x)	P("a")	{"a"/x}
R(f(y),z)	R(f("a"),x)	{"a"/y", x/z}
P(x)	P("b") || Q("a")	{"b"/x}

It is important to note that substitution can only be made for the variables in the subsuming clause. Variables in the subsumed clause can be replaced by new distinct constants before unification is attempted to avoid incorrect substitutions.

In this thesis, I will frequently use the phrase "partial subsumption" and a "partial constraint". Partial subsumption is explained here, and a partial constraint in the next section. Given the following clauses:

(1) P(x) || Q("a") (2) P("b") || R("b")

Clause 1 will "partially subsume" clause 2 by using the substitution {"b"/x}. Q("a") will remain from the subsuming clause, 1, after the partial subsumption and R("b") will be left from clause 2.

2.1.5. Integrity Constraints

In the usual AI or database definition, an integrity constraint is a formula or condition that cannot be violated if the database is to be consistent. As used in this thesis, integrity constraint is used to signify a condition that can never be true or perhaps that may be true, but is uninteresting and not a desired answer to a query or set of queries. In this manner, negated data or a headless Horn clause can be used to eliminate certain paths in the goal tree. Note, that in this situation, adding the constraint to a consistent database could make it inconsistent.

Kohli [Kohli 1983] presented a theory for directing a logic program using integrity constraints. This is that given a database of assertions and axioms; a database of constraints; and a goal to solve, integrity constraints can

be used to prune the goal tree. Perhaps, we can discover an unsolvable path and prune it before a lengthy search ultimately fails. Similarly, integrity constraints could be used to prevent finding some solutions that are not desired. If a constraint can subsume the goal clause or any subgoal, then the goal or subgoal is unsolvable or any answer it produces is not wanted. An example of the use of an integrity constraint in this manner is shown in Figure 1.

Futo [Futo 1984] developed the idea that rather than test an entire goal node for full subsumption, one could keep track of a sequence of partial subsumptions that could lead to failure. This would be particularly useful in keeping communication traffic low as would be desirable in PRISM.

Integrity constraints perhaps would be most useful in database applications where lengthy searches of data on secondary storage are required. Also, they may be useful in preventing impossible queries such as find Parent(x,x). Such facilities are not usually available in logic programming systems.

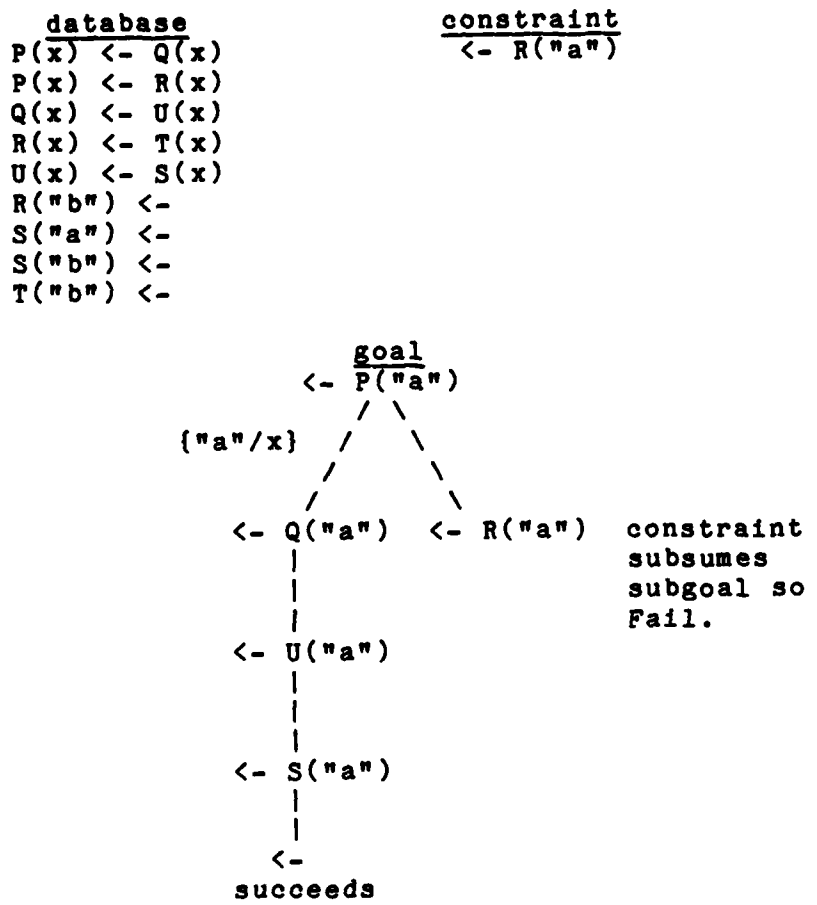

Fig. 1. Example of constraint pruning goal tree branch.

2.2 ZMOB

ZMOB is a microprocessor based parallel computer system under development at the University of Maryland [Rieger 1980, 1981a,b]. For complete details, the reader is referred to the references. Contained here is only a brief description of ZMOB and the features that have influenced the design and development of PRISM and the Constraint Machine.

The fully developed ZMOB will consist of 256 Z-80A microprocessors. Each Z-80A has 64K bytes of internal memory and no secondary storage. Each microprocessor is connected to a circular high-speed shift register (called the "conveyor belt") by its mail stop hardware. The conveyor belt consists of 257 bins, each 48 bits wide that "circulate" around the belt. A processor can read any other processor's bin, including its own, but can only fill its own bin when it "arrives" once each revolution. The 48 bits consist of 16 bits of data, 12 bits of destination address, 12 bits for the sender's address and 8 bits for control information.

2.2.1. Receiving Messages

Each processor has control over its own mailstop. By setting the control registers, it can elect to receive messages or block them out. There are two basic modes for receiving messages. The first is by address and the second

is by pattern. The processor can enable either or both modes and if by pattern, set the pattern it wishes to receive. The processor's address is fixed by the mailstop hardware. One final modification can be made to these two modes. A processor can set its "exclusive source" bit and in this mode it will only receive messages from the desired sender. This is useful for sending and receiving large blocks of data in an uninterrupted manner.

2.2.2. Sending Messages

When a processor sends a message, it also has the option of sending the message by pattern or address. In addition, it can elect to send it to one or all destinations. In the single destination mode, the message circulates around the belt until it arrives at its destination (if by address) or the first matching pattern. If not consumed by a receiver or the sender, the message will continue to circulate indefinitely. The receiver's mailstop removes the message and generates an inbound interrupt to its processor. This empties the sender's bin allowing him to send another message. Until the receiver reads the message in its mailstop inbound registers, it can not receive another message. When the sender's empty bin arrives back at its own mailstop, if there is a message to send, it is injected into the bin and an outbound interrupt generated to notify the processor that the next message is on its way. If the message is sent to all destinations, the only difference is

that the bin is not emptied by any of the receivers. The sender must do a "readback" of its own message to empty its bin prior to sending further messages. A "readback" must also be done to a single destination message that is not received by the destination. It is important to note that an "all destinations" message is not received by any processor whose mailstop is not enabled or enabled in a different mode. The sender can elect to let an unconsumed message circulate around the belt one or more times by setting his "readback" status.

2.2.3. ZMOB Host

The 257th mailstop is to be used for the host machine, a VAX 11/780. The host is the user's interface to the ZMOB. It can communicate over the belt just like any other processor on the belt. However, if it desires, the host can send messages in a non-maskable manner. The processors are guaranteed to receive a message sent out in this mode, regardless of their mailstop status.

2.3 PRISM.

PRISM is a parallel inference system designed and constrained to take advantage of the inherent parallelism of logic programming and the ZMOB architecture [Eisinger 1981] [Minker 1982][Kasif 1983]. In this system, the logical specification of the program and the control mechanism are separated. The system is to be distributed over

microprocessors of the ZMOB. There are currently three types of machines, the Problem Solvers (PSM), Intensional Database (IDB) and the Extensional Database (EDB) machines. This division is partly dictated by the size of the processors of the ZMOB, the nature of the algorithms used in unification and the anticipated size ot the databases. A VAX 11/780 and a PRISM Host process act as the user interface to the ZMOB and is referred to as the Host. For complete details, the reader is referred to the references.

2.3.1. The Problem Solving Machines (PSM).

The PSM is the heart of the PRISM system. Its primary task is to maintain the goal tree. In operation, a PSM receives a goal clause from the Host process. This becomes the root of a tree of goal clauses. In order to solve the problem, the PSM selects an atom of the clause for expansion. Selection is guided by heuristics and user supplied control structures. Depending on whether the selected atom is in an EDB or IDB, it is sent to one of these processes. If the selected atom unifies with either a procedure head or ground assertion, the goal clause is expanded with the procedure body and/or modified with the unifying substitution.

At any time, a goal node can be in one of four states: active, open, failure, or the empty clause. Active means that the goal has been selected for expansion, but has not been fully expanded. Open means that the clause has not

been selected for expansion. A failure node is unsolvable and an empty clause has been solved. While the PSM is waiting for answers from one of the database machines, or another PSM, it can work on other subgoal nodes in the goal tree.

When a PSM receives a goal clause, it begins to solve it. If a selected atom unifies with more than one procedure head, then two or more separate branches or subproblems are generated. These are called OR branches and can be solved independently of each other. The PSM can try to solve all subgoals or send one or more out to other free PSMs to solve. The subgoal in another, child PSM, becomes the root of a goal tree just like the original goal from the Host to the parent PSM. When a child PSM either solves its goal or fails, it reports this information back to its parent PSM. The original PSM reports its answers back to the Host process.

2.3.2. Intensional Database (IDB).

The intensional database consists of all of the procedures or axioms of the logic program. Since the procedures in the IDBs can contain functions, the unification algorithm must have an occur check. This is to prevent a substitution of a function for a variable if the function contains the same variable as a term. In most database applications, it would be expected that the database of

axioms would be relatively small compared to the extension. The decision to separate the intension from the extension was made because of the differences in the unification algorithms and the relative size of the two databases. In addition, the target machine, the ZMOB requires fairly small programs and data space.

In operation, the IDBs await requests from the PSMs. A PSM sends an atom it wishes to expand to an available IDB. There can be many identical IDB machines. The IDB matches the atom against all procedure heads in its database. If the unification succeeds with one or more procedures, the IDB sends a SUCCEED message to the PSM, else it sends a FAIL message. It also stores the answers which are the appropriately substituted procedure bodies and substitution lists. In this way, the IDB acts as a temporary buffer for the PSMs. Upon demand from the PSM, the IDB will send one or all of the answers back to the PSM. Any of the identical IDBs can service a new request from a PSM. Therefore, a PSM may have many IDBs working for it at once.

2.3.3. Extensional Database (EDB).

The extensional database consists of function free ground assertions. This means there are no variables or functions as terms in the assertions stored in the EDB machines. Each EDB contains different information, in contrast to the IDBs. A simple matching and substitution unif-

ication algorithm can be used. A single EDB machine can contain up to 16 relation tables and there can be multiple EDB machines. This gives a relational database capability of considerable size when the number of processors in the ZMOB is considered. The PSMs and EDBs make use of pattern addressed MATCH messages to identify which EDB contains the desired relation.

In operation, the EDB behaves similarly to the IDB. A request from the PSM consists of a literal with or without variables. If the unification succeeds, the answers are sent back to the PSM on demand. Answers take the form of a SUCCEED message for a fully instantiated query or one or more substitution lists for a query with variables.

2.3.4. Host - user Interface.

The VAX 11/780 and the PRISM Host process are to allow the user to interact with the problem solving system on ZMOB. The facilities of the VAX allow the user to create his EDB and IDB databases. Next using the host, the configuration is selected. This is how many IDBs to use and how the relations will be distributed in the EDBs. Loading of the ZMOB is accomplished by the Host after the databases are compiled by database compilers for each type of machine. When loading is complete, user queries are sent from the host to the ZMOB and answers returned to the Host. The Host program provides the user with utility commands that he can

give to the PRISM system for debugging and statistical purposes.

3. THE CONSTRAINT MACHINE COMPILER (CMC).

In the PRISM system on ZMOB, space will be very limited. All databases are first compiled into a compact internal representation with all variables, functions and predicates represented as integers. Strings are stored centrally in a string table and string terms are represented by pointers into the table. The compilation step creates this representation and also performs syntactic checking of the user's database prior to loading.

3.1. Constraint Database.

The input to the constraint machine compiler is a database of constraints representing negated data. An example of a small constraint database is given below:

```
(1)  <- Parent(x,x).
(2)  <- Grandfather(x,y),Female(x).
(3)  <- Grandfather("chip",x).
(4)  <- Married("karen",y).
```

The grammar for the constraint database is shown in Appendix A. The meaning of the constraints is fairly straight forward. The first says that no x is his own parent, the second that no grandfather is female, the third that "chip" is not grandfather of anyone and the fourth that "karen" is not married to anyone.

In addition to the database of constraints, the compiler needs a PRISM tag file. The tag file contains output from the IDB and EDB compilers. In it are all the predicate names, function names, their mapping to internal integer representation and the location of predicates (IDB, EDB, both or built in).

As constraints are parsed, the predicates are looked up in the tag file table (internal representation of the tag file) and converted to the internal integer value. In case an undefined predicate is encountered in a constraint, the entire constraint is discarded and an appropriate message printed. A constraint that contains a predicate found nowhere in the database is either an error or can never be violated so might as well be discarded.

Each predicate encountered is entered into a predicate table kept sorted by internal predicate name. The fact that a predicate occurs in the current input constraint is also recorded. This is done to build an index for fast look-up of predicates during operation of the Constraint Machine.

Terms in a literal are handled in a way similar to all terms in the PRISM system. Each term is tagged as to type: FUNCTION, STRING, NUMBER or CM_VARIABLE. Variables are assigned integer values that are unique for a given input constraint. In the case of strings and numbers, these are handled differently than elsewhere in the PRISM system. For

example, given the constraint 3 above, the following is generated in corresponding internal form:

```
Grandfather (x0,x1), EQ(x0,"chip").
```

In this form, if a partial subsumption takes place with an IDB body such as:

```
Grandfather(y0,y1).
```

the result will be a partial constraint of the form:

```
<- EQ(y0,"chip").
```

Subsequently, if a substitution of {"chip"/y0} is made, the predicate evaluates to true and the constraint will subsume the goal node. More will be said about this decision in Section 5. Numbers are handled in a manner similar to strings.

3.2 Output of the Compiler.

During the parsing of the input constraint file, only syntax errors and undefined predicates are reported. The output from the CMC is a file of binary coded data suitable for the CM to read and create its data structures. First in the file is the string table. The string table contains packed characters of all the strings encountered in the constraints. Representation of a string in a term consists of a length and offset into the vector of characters.

Following the string table is the predicate index. This is nothing more than the integer predicate name followed by a list of constraints that contain the predicate. Constraints are identified by their encounter order during the parsing phase. Finally, are the constraints themselves, represented as PRISM literal lists.

In the VAX implementation, of the CM, the output of the CMC is read in from a file. When PRISM is moved to the ZMOB, the CMC output file will be suitable for loading as a block into memory when the program is loaded and instructing the CM to begin reading at this location. Otherwise, the CM could read this data over the belt with the "over-the-belt" loader.

3.3 Current Implementation of the Compiler.

The Constraint compiler is written in C using the YACC compiler generator [Johnson 1978]. The code for the compiler is reproduced in Appendix B.

4. THE CONSTRAINT MACHINE (CM).

The Constraint Machine (CM) is the program that runs as a separate process in the VAX simulated version of PRISM. It will be a separate processor in the ZMOB implementation. Since the PRISM system was implemented without the CM, its design was constrained to disrupt the current system as little as possible. Therefore, the CM stores more data and communicates less with the PSM than might be considered optimal. The CM is an experimental part of the PRISM system and will be incorporated only if it demonstrates its utility in the VAX simulated version. More is said about making the CM an integral part of PRISM in Section 5.

4.1. Data Organization in the CM.

4.1.1. The Predicate Index and Constraint Table.

The predicate index is created at compile time by the CMC and read in when the CM initializes. It is stored as an array of predicate names and pointers to lists of constraint identifiers that contain the predicate. The predicate names are sorted so that if the number of predicates becomes large, a fast binary search can be used to rapidly find all constraints that could possibly be violated by a given PSM input.

The constraint table is an array of pointers to the literal lists that compose the constraints. The constraint

identifier is the array index used to find a pointer to the constraint literal list. The combination of the predicate index and constraint table provide for reasonably fast retrieval of constraints that may be violated in the database of constraints. Storage for these tables is allocated from the heap at the time the compiled database is read.

4.1.2. The Partial Constraint Tree.

The constraint machine must keep track of all partial subsumptions that occur for a given PSM goal node. Descendant nodes of a PSM node where a partial subsumption occurred must be checked against this new constraint or partial constraint as well as the database of original constraints. This is because the PSM will not send the entire node to the CM, but just send the body and substitution that are replacing or modifying the expanded literal. The partial constraint tree keeps track of partial subsumptions due to other literals that comprise the entire goal node in the PSM.

The CM is designed to be able to handle multiple PSMs and multiple queries from each PSM. Within a given query, no data is stored unless a partial subsumption occurs. From that point on, a corresponding partial tree node is created for each descendant of the PSM node. Therefore, for a given PSM goal tree, the CM may have a forest of subtrees, the root of each being a PSM subgoal node where a partial sub-

sumption occurred.

The number of PSMs and queries within a PSM is not anticipated to be large. However, the number of goal tree nodes within a query may be quite large. Therefore, the corresponding number of nodes in the partial tree may be large. Locating a new PSM node's parent node then becomes a problem. To solve this problem, an array of pointers is allocated with each query node. The PSM parent node number, modulo the size of the pointer array, is used as an index into the array of pointers, giving a pointer to the partial tree node. A check of the identified node's node number positively identifies the correct node. If the pointer in the array is null, then the parent does not exist. If a different node is in the query index, then the parent pointer may have been over written by a more recent node. In this case, a search must be undertaken to see if the parent exists in the node list. Thus the array is like a cache of pointers to recent nodes. Figure 2 shows an example of this data structure. As an example, if a CHECKSONST message has parent node 100, then the index 100 into the array of pointers gives a pointer to the parent, 100.

Each node has a pointer to its parent. This pointer will allow a trace back from a node where a violation occurred to identify the offending expansions and substitutions leading to the constraint violation.

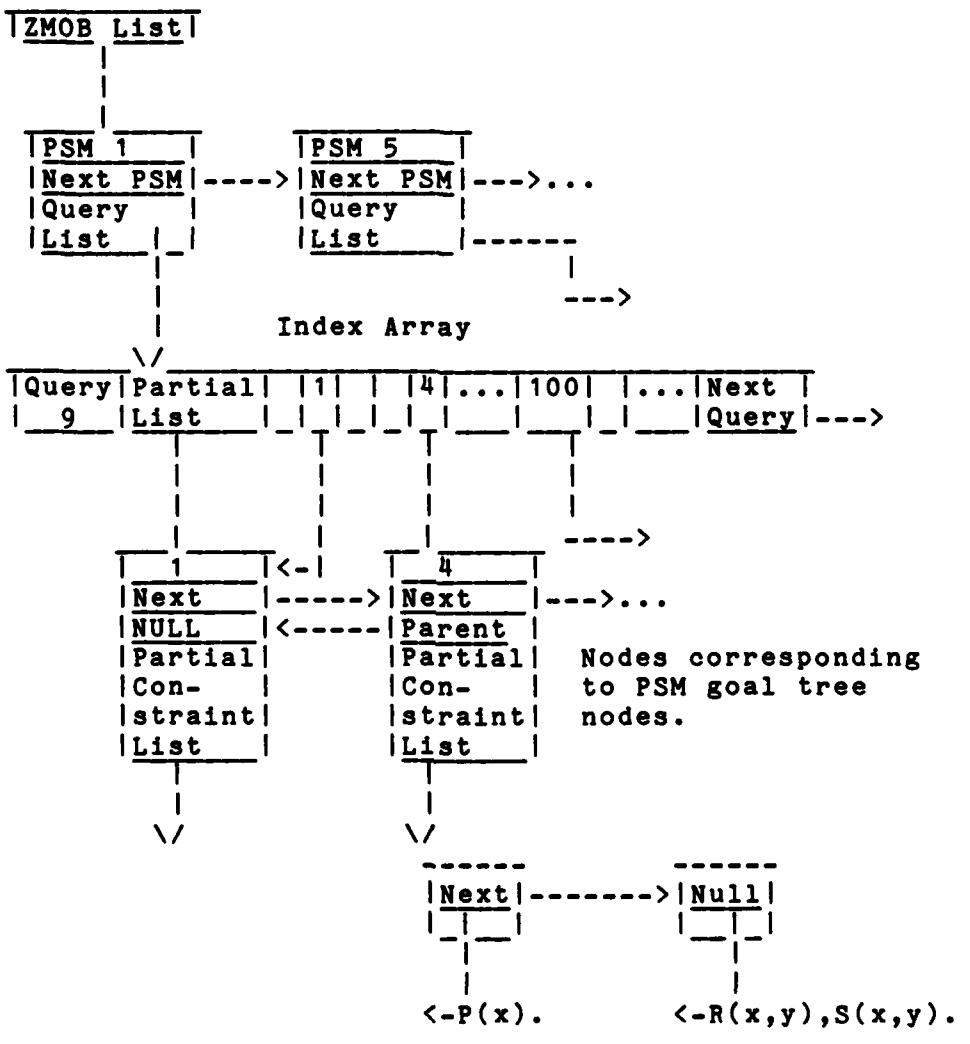

Fig. 2. Structure of the partial constraint tree.

Within each node in the partial tree is kept the PSM's node number and a pointer to a partial constraint list. The partial constraint list contains all partial constraints that are applicable to descendants of this node.

4.2. Communication.

The major communication of the CM is with the PSMs. However, Host messages for certain debugging and statistics are also handled.

4.2.1. Communication PSM and CM.

(1) CHECKCONST:

This message is used by the PSM to send a new query, expansion body or substitution list to the CM. All CHECKCONST messages contain the PSM new node number, parent node number and a body and substitution. Either the body or the substitution list may be empty. In the case of a query, the message contains the query literal list and the substitution list is empty. When a literal is expanded with an IDB body, the IDB body literal list and unifying substitution list are in the message. In the case of an EDB expansion, the body is empty and only a substitution list is received by the CM.

(2) ERASE:

The ERASE message is sent by a PSM whenever it ter-

minates a query, either successfully, due to failure or due to a command from the Host. The CM responds by removing the indicated query and all partial constraint nodes from the partial tree. If the PSM has no other queries active, the PSM entry is also deleted.

(3) VIOLATION:

This is the message returned to the PSM when a constraint fully subsumes a goal tree node. The violation may be the result of a full subsumption from one CHECKCONST message or as a series of partial subsumptions occurring in ancestors and ending in the current node. The information sent to the PSM is the query and node number of the violating goal node.

4.2.2. Communication CM - Host.

(1) TERMINATE:

When this message is received from the Host, the CM terminates execution.

(2) ASK_DUMP:

This command is used for debugging. When this command is received by the CM, it responds with a dump of the entire partials tree. The predicate index and constraint table can also be dumped.

(3) DUMP:

This message is the reply to the ASK_DUMP message.

(4) SETSTATISTICS:

This message is sent by the Host to the CM. The response is to start gathering statistics as specified in the message header.

(5) STATISTICSOFF:

This message is used by the Host to stop the collection of statistics.

(6) GETSTATISTICS:

This message is sent by the Host to the CM. The CM responds by sending to the Host the requested statistics. Receipt of this message without a previous SETSTATISTICS is an error.

4.3. Operation of the CM.

4.3.1. Working Cycle of the CM.

The constraint machine working cycle begins with receipt of a CHECKCONST message from a PSM. The PSM number, query number and parent node number are used to determine if the parent node is in the partial tree. If so, the parent's partial constraints are retrieved. The substitution list in the message is applied to the retrieved partial constraints. All equality predicates that are fully instantiated are evaluated and discarded if true and the partial constraint is discarded if false. If by removing true equality predicates, the partial constraint becomes empty, then a full

subsumption has occurred and the new PSM node is a failure node. A violation message is sent to the PSM and the new node deleted.

If there are still partial constraints and the new PSM node's body is not empty, then a check is made for subsumptions of the new body by the parent node's partial constraints. If a further partial subsumption takes place, the parent's partial constraint is replaced by the new shorter partial constraint on the child node's list. The subsuming literals are deleted and appropriate substitutions made. Unchanged parent's partial constraints are kept intact to check the new node's descendants. If a full subsumption takes place between the parent's partial constraint and the new body, a VIOLATION message is sent to the PSM and the new node deleted.

Now, if the new node is still not a failure node, the new body must be checked for subsumption by the constraints in the original database. The predicates in the new body are looked up in the predicate index to get a list of all constraints that could be violated. These constraints are then input with the body to the subsumption algorithm. If a new partial subsumption occurs, the new node is created in the partials tree if necessary, and the new partial constraints added to its list. Any fully instantiated equality predicates in a subsumption are properly handled as outlined before.

4.3.2. Subsumption Algorithm Implementation.

The PSM sends the entire initial query literal list to the CM. Subsequently, as each literal in the query is expanded by an IDB body or instantiated by an EDB substitution, the substituted body and/or substitution list is sent to the CM. The remaining literals in the PSM goal node are not sent again. Since the full node is never sent to the CM, except for the root query node, it is very likely that only partial subsumptions will occur with any given PSM body and substitution list. The CM keeps track of all partial subsumptions that result from a body and constraint (or partial constraint). There may be more than one partial constraint resulting from partial subsumption of a goal by a constraint. For example:

constraint	body
<- P(x), R(x),S(y)	P("a"),R("b")

gives rise to two partial constraints:

<- P("b"),S(y)

<- R("a"),S(y)

Successor nodes of the node containing the body P("a"),R("b") must be checked against both partial constraints. Of course, this makes the algorithm that much more expensive.

In operation, the subsumption algorithm receives a constraint and body literal list. It checks for matching predicates. Any pairs that match result in a call to the unification algorithm. If the literals are unifiable, the resulting substitution is applied to the body and constraint after the subsuming and subsumed literals have been deleted. The resulting shortened body and constraint are then checked for further matching literals and subsumptions until no further subsumption takes place, the body becomes the empty list or the constraint becomes empty. If the constraint becomes empty then the constraint has fully subsumed the goal body and a violation results.

Once all subsumptions have been generated, each is evaluated. This process looks for equality predicates in the partial constraint. If there are any that are fully instantiated, then one of two possibilities exist. If true, then the literal is discarded. If by doing so, the constraint becomes empty, then a full subsumption has occurred. If the equality predicate is false, then this constraint cannot be violated and is discarded. The subsumption algorithm returns a list of partial subsumptions and one of 4 status flags: okay, partial, full or no violation possible.

The unification algorithm is a slightly modified version of the PRISM IDB unification algorithm. In PRISM, each term is tagged with a type: STRING, NUMBER, FUNCTION, IDB VARIABLE, CM VARIABLE or PSM VARIABLE. Initially, all

variables in the constraint database are tagged as CM VARIABLES by the compiler. The unification algorithm is biased to consider PSM variables as constants and not allow substitutions to be made for them. This has the effect of changing all variables in a body to distinct constants as required in the Chang and Lee algorithm [Chang 1973]. However, during the unification and substitution process, PSM VARIABLES are substituted for CM VARIABLES in the constraints. Then, they are only unifiable with themselves or they can be changed to other terms if they appear in a substitution list from the PSM.

4.4. Current Implementation of the CM.

The current implementation of the CM is written in the C language and the code appears in Appendix C. All storage is allocated and deallocated from a heap. It has been tested by reading the compiled constraints and input PSM messages from files. Likewise, the output violation messages are written to a file.

The ASK_DUMP message is only implemented when running in the VAXDEBUG mode. An ASK_DUMP message causes a dump of the predicate index, constraint table and partial constraint tree to the debug file. There is no DUMP message sent in reply. No collection of statistics has yet been implemented.

The code contains the necessary options and calls to PRISM library routines to operate with the PRISM belt simulator using the PRISM ipc i/o routines. Since the VAX has changed to Berkeley 4.2, these routines are no longer supported. In order to compile and run, this code has been made into comments. When the PRISM i/o library is rewritten, there will be a few modifications needed as well as removing comment delimiters from the i/o code.

The current PSM does not know about the CM. It will have to be modified to send all queries and IDB/EDB expansions to the CM. Also, the PSM will need to be able to handle the return VIOLATION messages from the CM. The CM expects to receive an ERASE message from the PSM at the conclusion of any query.

5. FUTURE CONSIDERATIONS.

The CM is an add on feature of the PRISM system. It provides a method to evaluate a theory that integrity constraints can be used to direct and limit the search of a logic programming system. Therefore, its value will have to be determined by experimentation. It is quite likely that the CM's utility will be very application dependent. It is easy to generate examples of both extremes.

In its present implementation, the CM can demonstrate its feasibility as part of the VAX PRISM system. Complete evaluation of the CM must wait until the PSM is modified to send and receive CM messages. The main criteria for evaluating the CM will be:

(1) The execution time of the PSM for queries with and without the CM in operation.

(2) The number of nodes generated and space consumed by the PSM for queries with and without the CM in operation.

Once the initial modifications are made and if the initial test are favorable, there are some reasonable modifications to be made to the whole PRISM system to take advantage of the CM.

5.1. When to Use the CM.

The CM should be an optional feature to the PSMs. That is, the decision whether to use the CM or not must be made at load time or later. If there is no database of constraints, then no CHECKCONST messages should be sent to the CM. Carrying this argument one step further, suppose a database of constraints exists, but an entire query's goal tree contains only predicates that are not in the constraint database. Again, sending CHECKCONST messages is of no value.

For the preceding reasons, a reasonable modification to PRISM would be to tag predicates and IDB bodies as to whether they contain predicates that exist in the CM database. This would necessitate a major modification of the current database compilers used on the VAX. The CMC should not run until the IDB and EDB compilers have defined the predicates in the database. But then the IDB and EDB cannot tag their bodies and predicates with the "contained in CM" tag. Therefore, the compilation could be changed into a two pass operation or all compilers could be combined into one.

Assuming tagged predicates and bodies, there is a further enhancement that could be made. Suppose there has been no partial subsumptions in any ancestor node in a PSM goal tree. Further, the selected expansion involves only predicates not contained in the constraint database. Again the CHECKCONST message is of no value and could be avoided. To do this would require a tag on PSM goal tree nodes to

indicate whether a partial subsumption has taken place in an ancestor node. In order for the PSM to be able to correctly maintain these tags, the CM would have to send a "partial subsumption" message to the PSM whenever one occurred. The tag would be propagated from that node down and each expansion sent to the CM. Even if the expansion involved only predicates not contained in the constraints, the substitution list could cause a further subsumption to occur.

5.2. Adding Constraints to PSM Nodes.

The current implementation of the CM compiler generates equality predicates for all constants in the constraint database. The equality predicates are placed at the end of the constraint literal list. During the subsumption algorithm, as constraint-literals subsume body-literals, the constraint shrinks. When only equality predicates are left, all that remains is to wait until they are fully instantiated and attempt evaluation. Both the CM and PSM are capable of evaluating equality predicates so why not send the list to the PSM. First, the equality predicates would be converted into inequality predicates. Then the PSM would add these constraints to its goal tree node. From then on, the PSM could determine if the constraint were violated. For example, the constraint:

```
<- P("john")
```

compiled into

```
<- P(x0), EQ(x0,"john")
```

would partially subsume the query

```
<- P(x)
```

leaving

```
EQ(x0,"john")
```

Converting this to

```
NE (x0,"john")
```

and appending this to a PSM node would cause a failure if the substitution of {"john"/x0} is made.

Implementation of this feature would require the necessary PSM and CM messages and minor modifications to both machines.

5.3. When a PSM Creates a Child PSM.

There is a PRISM system problem that must be solved in order for the CM to become an integral part of PRISM. That is, what should be the relationship between the PSMs and CMs. Should it be one-to-one or one CM for each PSM query? Then whatever approach is taken, there is the problem of what to do when a PSM sends a subgoal to another PSM. In this case, the new PSM could either send to the parent PSM's CM. A more reasonable alternative would be for the child PSM's CM to get a copy of any partial constraints that apply

to the new root goal node from the parent PSM's CM. Solution to this problem is an entire PRISM system problem and need not be addressed unless the CM seems a useful addition on a permanent basis.

6. SUMMARY.

A Constraint Machine and associated compiler have been implemented in C for the PRISM VAX simulator system. The CM is operational and capable of extensive testing to determine its utility in a logic programming system. The CM uses a database of constraints to identify failure nodes in a PSMs goal tree. It is hoped that this system can improve the execution of logic programs in the PRISM system. At least, it will be able to test the theory that constraints can be used successfully in this manner.

Appendix A - Constraint File Grammar

This grammar describes the user supplied database of integrity constraints.

<file> ::= <constraint list>

<constraint list> ::= <constraint> <constraint list> | E

<constraint> ::= <- <literal list> .

<literal list> ::= <predicate literal> |
<predicate literal> <literal list>

<predicate literal> ::= <predicate name> (<argument list>) |
<predicate name> ( )

<predicate name> ::= <identifier>

<argument list> ::= <argument> | <argument> , <argument list>

<argument> ::= <variable> | <constant> | <function>

<variable> ::= <identifier>

<constant> ::= <number> | <string>

<function> ::= <function name> (<argument list>) |
<function name>()

<function name> ::= <identifier>

<string> ::= <double quote> <char list> <double quote>

<char list> ::= <char> | <char> <char list>

<char> ::= <non double quote char>

<double quote> ::= "

<non double quote char> ::= any character except "

APPENDIX B

Appendix B contains the C code source listing of the Yacc grammar and routines that comprise the Constraint Machine compiler.

```
/* File cmc.h: This file contains the global data structure
   types used in the Constraint Machine Compiler          */

/* # include  "prism.h"  removed to get rid of ipc stuff */
# include   "tempprism.h"

/*-------------------------------------------------------*/

typedef enum
    {
        UNKNOWN_TAG_TYPE,
        PRED_NAME,
        FUNC_NAME
    }
    TAG_TYPE;

/*-------------------------------------------------------*/
/* String table                                          */
/*-------------------------------------------------------*/

typedef struct string_item
    {
        STRING_DESCRIPTOR   String;
        unsigned int        Offset;
        struct string_item  *Next_String;
    }
    STRING_TABLE_ITEM;

/*-------------------------------------------------------*/
/* Constraint list                                       */
/*-------------------------------------------------------*/

typedef int                         CONSTRAINT_NAME;

typedef struct constraint_list
    {
        LITERAL                     *Literal_List;
        struct constraint_list      *Next_Constraint;
    }
    CONSTRAINT_LIST;
```

```
/*-------------------------------------------------------------*/
/* Predicate index list                                        */
/*-------------------------------------------------------------*/

typedef struct contained_in_list
    {
        CONSTRAINT_NAME              Constraint_Id;
        struct contained_in_list     *Next_Constraint;
    }
    CONTAINING_CONSTRAINTS;

typedef struct predicate_name_list
    {
        LITERAL_NAME                 Name;
        struct predicate_name_list   *Next_Predicate;
        CONTAINING_CONSTRAINTS       *Possible_Constraint;
    }
    PREDICATE_LIST;
```

```
/* File cm.c: This file contains the main routine for
   the Constraint Machine Compiler                          */

#include "cmc.h"

main(argc,argv)
    int     argc;
    char    *argv[];
{
    Init_Debug();
    Init_Heap();
    Init_Variable_Table();
    Get_Options(argc,argv);
    yyparse();
}
/*-------------------------------------------------------------*/
```

```
/* File grammar.y: Containts the Yacc grammar for the
   Constraint Machine compiler.  The length of strings
   limited to 20 characters because of the limitations
   of the PRISM library routines in variable.c.
   They should be changed to allow variable
   names to be up to 200 characters long like strings  */
%{
#include "cmc.h"
%}
%{
    extern  FILE                *debugf;
    extern  int                 Line_Number;
    extern  LITERAL             *Alloc_LITERAL();
    extern  TERM                *Alloc_TERM();
    extern  char                *Alloc_CHAR();
    extern  TAG_TABLE           Tag_File_Table[];
    extern  int                 Update_Tag_Table();
    extern  STRING_TABLE_ITEM   *String_Table;
    extern  FILE                *Tag_File;
    extern  BOOLEAN             All_Predicates_Defined;
    extern  LITERAL             *Equals_List;
    extern                      itoa();

/*-------------------------------------------------------------*/
    int                 Tag_Entry_Index;
    char                Pred_Name[MAX_IDENTIFIER_LENGTH+1];
    LITERAL             *temp_lit;
    TERM                *temp_term;
    char                dummy_variable_name[200];

%}
%union
    {
        int                     stval_int;
        char                    *stval_chptr;
        BOOLEAN                 stval_bool;
        LITERAL                 *stval_literal;
        CM_STRING_DESCRIPTOR    stval_string;
        TERM                    *stval_term;
        TAG_TABLE               stval_tag;
    }
%token <stval_int>
    NUMBER DEFAULT LR RANDOM
    ILLEGAL_SYMBOL ARROW
%token <stval_chptr>
    ID
%token  <stval_string>
    STRING_T
%type <stval_term>
    arg arg.list arg.list.with.parens func.name
%type   <stval_literal>
    literal literal.list
```

```
%start file

%%

file                    : constraint.list
    {
        Write_CM_File();
    };


constraint.list             : constraint constraint.list
                    | constraint;

constraint                  : ARROW literal.list '.'
    {
        if (All_Predicates_Defined) then
        {
            Add_Literal_To_Literal_List(Equals_List,&$2);
            Insert_Constraint_Into_List($2);
        }
        Equals_List = NULL;
        Init_Variable_Table();
        All_Predicates_Defined = TRUE;
    };

literal.list                : literal ',' literal.list
    {
        $1->Next_Literal = $3;
        $$ = $1;
    }
                    | literal
    {
        $$ = $1;
    };


literal                 : identifier arg.list.with.parens
    {
        $$ = Alloc_LITERAL();
        $$-> Next_Literal = NULL;
        $$->Arg_List = $2;
        Tag_Entry_Index = Find_Procedure_Name(Pred_Name,
                            Tag_File_Table,UNDEFINED_TAG);
        if (Tag_Entry_Index == -1)
        {
            fprintf(stderr,
                "Line %d: Undefined Literal %s",Line_Number,
                                                Pred_Name);
            fprintf(stderr," constraint discarded0),
            All_Predicates_Defined = FALSE;

        }
```

```
        $$->Name =
            Tag_File_Table[Tag_Entry_Index].Encoded_Name;
        Add_To_Predicate_List($$->Name);
        $$->Location =
                Tag_File_Table[Tag_Entry_Index].Location;
    }
                        | error ')'
    {
        fprintf(debugf,"error in literal0);
    };

identifier              : ID
    {
        strcpy(Pred_Name,$1);
    };

arg.list.with.parens    : '(' arg.list ')'
    {
        $$ = $2;
    }
                        | '(' ')'
    {
        $$ = NULL;
    };

arg.list                : arg ',' arg.list
    {
        $1->Next_Term = $3;
        $$ = $1;
    }
                        | arg
    {
        $$ = $1;
    };

arg                     : STRING_T
    {
        $$ = Alloc_TERM();
        $$->Type = CM_VARIABLE;
        Get_String_Value_Given_Offset(dummy_variable_name +
                                                1,$1.Offset);
        dummy_variable_name[0] = '_';
        $$->Value.Variable.Name =
                Assign_Variable_Number(dummy_variable_name);
        $$->Value.Variable.Answer_Variable = FALSE;
        $$->Next_Term = NULL;
                            /* now create a EQ literal */
        temp_lit = Alloc_LITERAL();
        temp_lit->Next_Literal = NULL;
        Tag_Entry_Index = Find_Procedure_Name("EQ",
                                            Tag_File_Table);
        temp_lit->Name =
                Tag_File_Table[Tag_Entry_Index].Encoded_Name;
```

```
    temp_lit->Location = BUILT_IN;
    temp_term = Alloc_TERM();
    temp_lit->Arg_List = temp_term;
    temp_term->Type = CM_VARIABLE;
    temp_term->Value.Variable.Name =
                                    $$->Value.Variable.Name;
    temp_term->Value.Variable.Answer_Variable = FALSE;
    temp_term->Next_Term = Alloc_TERM();
    temp_term = temp_term->Next_Term;
    temp_term->Type = CM_STRING;
    temp_term->Value.CM_String = $1;
    temp_term->Next_Term = NULL;
    Add_Literal_To_Literal_List(temp_lit,&Equals_List);
}
                    | ID
{
    $$ = Alloc_TERM();
    $$->Type = CM_VARIABLE;
    $$->Value.Variable.Name = Assign_Variable_Number($1);
    $$->Value.Variable.Answer_Variable = FALSE;
    $$->Next_Term = NULL;
}
                    | NUMBER
{
    $$ = Alloc_TERM();
    $$->Type = CM_VARIABLE;
    itoa($1,dummy_variable_name + 1);
    dummy_variable_name[0] = '_';
    $$->Value.Variable.Name =
          Assign_Variable_Number(dummy_variable_name);
    $$->Value.Variable.Answer_Variable = FALSE;
    $$->Next_Term = NULL;
                        /* now create a EQ literal */
    temp_lit = Alloc_LITERAL();
    temp_lit->Next_Literal = NULL;
    Tag_Entry_Index =
                Find_Procedure_Name("EQ",Tag_File_Table);
    temp_lit->Name =
          Tag_File_Table[Tag_Entry_Index].Encoded_Name;
    temp_lit->Location = BUILT_IN;
    temp_term = Alloc_TERM();
    temp_lit->Arg_List = temp_term;
    temp_term->Type = CM_VARIABLE;
    temp_term->Value.Variable.Name =
                                    $$->Value.Variable.Name;
    temp_term->Value.Variable.Answer_Variable = FALSE;
    temp_term->Next_Term = Alloc_TERM();
    temp_term = temp_term->Next_Term;
    temp_term->Type = INTEGER;
    temp_term->Value.Integer.Value = $1;
    temp_term->Next_Term = NULL;
    Add_Literal_To_Literal_List(temp_lit,&Equals_List);
}
```

```
                        | func.name '(' arg.list ')'
    {
        $1->Value.Function.First_Arg = (int *) $3;
        $1->Next_Term = NULL;
        $$ = $1;
    };

func.name               : ID
    {
        $$ = Alloc_TERM();
        $$->Type = FUNCTION;
        Tag_Entry_Index = Find_Procedure_Name($1,
                                    Tag_File_Table,FUNCTION);
        if (Tag_Entry_Index == -1)
        {
            fprintf(stderr,"Line %d: Unknown Function %s0,
                                            Line_Number,$1);
        }
        $$->Value.Function.Name =
                Tag_File_Table[Tag_Entry_Index].Encoded_Name;
    };
%%
/*-------------------------------------------------------*/
```

```
/* File globals.c: This file contains all global variables
   used by more than one file in the Constraint Machine
   compiler                                                */

#include "cmc.h"

int                 Line_Number = 1,
                    String_Table_Size = 0,
                    Number_Predicates = 0;
CONSTRAINT_NAME     Constraint_Number = 1;

TAG_TABLE           Tag_File_Table[MAX_PREDICATES];

FILE                *Axiom_File =   stdin,
                    *debugf     =   stderr,
                    *CM_File,
                    *Tag_File,
                    *EDB_Tag_File;

STRING_TABLE_ITEM   *String_Table;

PREDICATE_LIST      Predicate_Table = {0, NULL, NULL};

CONSTRAINT_LIST     *Constraint_Table = NULL;

BOOLEAN             Syntax_Only = FALSE;
BOOLEAN             Output_Listing = FALSE;
BOOLEAN             All_Predicates_Defined = TRUE;
LITERAL             *Equals_List = NULL;
/*-------------------------------------------------------------*/
```

```
/* File actions.c: This file contains the parser actions
   for the Constraint Machine Compiler                    */

#ifdef CMCDEBUG
#define ACTIONSDEBUG
#endif

#include "cmc.h"

extern                              Print_LITERAL_List();
extern  FILE                        *debugf;
extern  FILE                        *CM_File;
extern  int                         Number_Of_Arguments();
extern  int                         Number_Predicates;
extern  PREDICATE_LIST              Predicate_Table;
extern  CONSTRAINT_LIST             *Constraint_Table;
extern  int                         Constraint_Number;
extern  CONTAINING_CONSTRAINTS
                            *Alloc_CONTAINING_CONSTRAINTS();
extern  PREDICATE_LIST              *Alloc_PREDICATE_LIST();
extern  CONSTRAINT_LIST             *Alloc_CONSTRAINT_LIST();

/*-------------------------------------------------------*/

static PREDICATE_LIST   *insert_new_predicate(predicate,
                                              front,
                                              back)
    LITERAL_NAME            predicate;
    PREDICATE_LIST          *front;
    PREDICATE_LIST          *back;
/* link node in as successor of back,
   New nodes next pointer set to front.
   increment count of predicates                      */
{
#   ifdef ACTIONSDEBUG
        fprintf(debugf,"Entering insert new predicate0);
#   endif
    back->Next_Predicate = Alloc_PREDICATE_LIST();
    back->Next_Predicate->Next_Predicate = front;
    back->Next_Predicate->Name = predicate;
    back->Next_Predicate->Possible_Constraint = NULL;
    Number_Predicates++;
    return(back->Next_Predicate);
}
/*-------------------------------------------------------*/

static  insert_constraint_in_predicate_list(predicate_list)
    PREDICATE_LIST          *predicate_list;

/* predicate list points to a node in the index
   ito which to add the current constraint */

{
```

```
    CONTAINING_CONSTRAINTS          *temp;

#   ifdef ACTIONSDEBUG
        fprintf(debugf,
            "Enter insert constraint in predicate list0);
#   endif
    if (predicate_list->Possible_Constraint == NULL) then
    {
        predicate_list->Possible_Constraint =
                            Alloc_CONTAINING_CONSTRAINTS();
        predicate_list->Possible_Constraint->Constraint_Id=
                                        Constraint_Number;
        predicate_list->Possible_Constraint->Next_Constraint
                                                = NULL;
        return;
    }
    else
    {
        temp = predicate_list->Possible_Constraint;
    }
    while ((temp->Next_Constraint != NULL) &&
            (temp->Constraint_Id != Constraint_Number))
        temp = temp->Next_Constraint;
    if (temp->Constraint_Id == Constraint_Number) return;
    if (temp->Next_Constraint == NULL) then
    {
        temp->Next_Constraint =
                        Alloc_CONTAINING_CONSTRAINTS();
        temp->Next_Constraint->Next_Constraint = NULL;
        temp->Next_Constraint->Constraint_Id =
                                        Constraint_Number;
    }
}
/*-------------------------------------------------------*/


Add_To_Predicate_List(predicate)
    LITERAL_NAME                predicate;

{
    PREDICATE_LIST              *temp;
    PREDICATE_LIST              *front;
    PREDICATE_LIST              *back;

#   ifdef ACTIONSDEBUG
        fprintf(debugf,"Enter add to predicate list 0);
#   endif
    front = &Predicate_Table;
    back = NULL;
    temp = NULL;
    while ((front->Name < predicate) &&
                        (front->Next_Predicate != NULL))
    {
```

```
        back = front;
        front = front->Next_Predicate;
    }
    if (front->Name == predicate) then
        temp = front;
    else if ((front->Name < predicate) &&
                    (front->Next_Predicate == NULL)) then
        temp = insert_new_predicate(predicate,NULL,
                                                back=front);
    else
        temp = insert_new_predicate(predicate,front,back);
    insert_constraint_in_predicate_list(temp);
}
/*-------------------------------------------------------*/

Insert_Constraint_Into_List(literal_list)
    LITERAL             *literal_list;
/* hook new list of literals (a constraint) to list of all
   constraints. Increment global count of constraints    */

{
    CONSTRAINT_LIST         *temp;

#   ifdef ACTIONSDEBUG
        fprintf(debugf,
                "Enter Insert Constraint Into List0);
#   endif
    temp = Constraint_Table;
    Constraint_Number++;
    if (temp == NULL) then
    {
#       ifdef ACTIONSDEBUG
            fprintf(debugf,
                "Constraint_Table NULL, Alloc ing0);
#       endif
        Constraint_Table = Alloc_CONSTRAINT_LIST();
        Constraint_Table->Next_Constraint = NULL;
        Constraint_Table->Literal_List = literal_list;
    }
    else
    {
    while (temp->Next_Constraint != NULL)
        temp = temp->Next_Constraint;
    temp->Next_Constraint = Alloc_CONSTRAINT_LIST();
    temp->Next_Constraint->Next_Constraint = NULL;
    temp->Next_Constraint->Literal_List = literal_list;
    }
#   ifdef ACTIONSDEBUG
        Print_LITERAL_List(literal_list);
        fprintf(debugf,
                    "Exit Insert Constraint Into List0);
#   endif
}
```

```
/* File alloc.c: This file contains the storage allocation
   routines used in the Constraint Machine Compiler      */

#ifdef  CMCDEBUG
#   define  ALLOCDEBUG
#endif

# include "cmc.h"

extern  FILE        *debugf;
extern  char        *alloc_block();

/*-------------------------------------------------------*/
#define STRING_TABLE_ITEM_SIZE
        ((sizeof(STRING_TABLE_ITEM) +
                                    sizeof(int)-1)/sizeof(int))

#define CONSTRAINT_LIST_SIZE
        ((sizeof(CONSTRAINT_LIST) +
                                    sizeof(int)-1)/sizeof(int))

#define PREDICATE_LIST_SIZE
        ((sizeof(PREDICATE_LIST) +
                                    sizeof(int)-1)/sizeof(int))

#define CONTAINING_CONSTRAINTS_SIZE
    ((sizeof(CONTAINING_CONSTRAINTS) +
                                    sizeof(int)-1)/sizeof(int))
/*-------------------------------------------------------*/

char *Local_Alloc(size)
    unsigned int    size;
{
    char    *ptr;

    ptr = alloc_block(&size);
    return(ptr);
}
/*-------------------------------------------------------*/

Local_Dealloc(ptr,size)
    char            *ptr;
    unsigned int    size;
{
    dealloc_block(ptr,size);
}
/*-------------------------------------------------------*/

STRING_TABLE_ITEM *Alloc_STRING_TABLE_ITEM()
{
    return((STRING_TABLE_ITEM *)
                        Local_Alloc(STRING_TABLE_ITEM_SIZE));
}
```

```
/*-------------------------------------------------------------*/

CONSTRAINT_LIST     *Alloc_CONSTRAINT_LIST()
{
    return((CONSTRAINT_LIST *)
                        Local_Alloc(CONSTRAINT_LIST_SIZE));
}
/*-------------------------------------------------------------*/

PREDICATE_LIST      *Alloc_PREDICATE_LIST()
{
    return((PREDICATE_LIST *)
                        Local_Alloc(PREDICATE_LIST_SIZE));
}
/*-------------------------------------------------------------*/

CONTAINING_CONSTRAINTS  *Alloc_CONTAINING_CONSTRAINTS()
{
    return((CONTAINING_CONSTRAINTS *)
                Local_Alloc(CONTAINING_CONSTRAINTS_SIZE));
}
/*-------------------------------------------------------------*/
```

```
/* File cmclib.c: This file contains the common routines
   for the Constraint Machine Compiler.  These routines
   are general and should be added to the PRISM library */

#ifdef CMCDEBUG
#   undef CMCLIBDEBUG
#endif

#include "cmc.h"

extern  FILE                            *debugf;
#ifdef CMCLIBDEBUG
extern                                  Print_LITERAL_List();
#endif
/*-------------------------------------------------------*/

Add_Literal_To_Literal_List(literal,literal_list)

    /* adds literal (or a literal list) to the end
                                of the first literal list */

    LITERAL                             *literal,
                                        **literal_list;
{
    LITERAL                             *temp;
#   ifdef CMCLIBDEBUG
        fprintf(debugf,"Adding_Literal_To_Literal_List0);
        Print_LITERAL_List(*literal_list);
        fprintf(debugf,"   now the literal0);
        Print_LITERAL_List(literal);
        fprintf(debugf,"0);
#   endif
    if (*literal_list == NULL) then
    {
        *literal_list = literal;
    }
    else
    {
    temp = *literal_list;
    while(temp->Next_Literal != NULL)
    {
        temp = temp->Next_Literal;
    }
    temp->Next_Literal = literal;
    }
}
/*-------------------------------------------------------*/

reverse(string)
    char                                string[];
{
    int                                 c, i, j;
```

```
#   ifdef CMCLIBDEBUG
        fprintf(debugf,"  Reversing string0);
#   endif
    for (i = 0, j = strlen(string) -1; i < j; i++, j--)
    {
        c = string[i];
        string[i] = string[j];
        string[j] = c;
    }
}
/*-------------------------------------------------------------*/

itoa(n,array)

    int                                 n;
    char                                array[];
{
    int                                 i, sign;

#   ifdef CMCLIBDEBUG
        fprintf(debugf,"  Converting integer to ASCII0);
#   endif
    if ((sign = n) < 0) then
        n = -n;
    i = 0;
    do  {
            array[i++] = n % 10 + '0';
        } while ((n /= 10) > 0);
    if (sign < 0)
        array[i++] = '-';
    array[i] = ' ';
    reverse(array);
}
/*-------------------------------------------------------------*/
```

```
/* File output.c: This file contains the routines that
   write the predicate index, string table, and the
   constraint list to the output file.  The output
   file is used by the CM to create the database
   of constraints and the predicate index              */

#ifdef CMCDEBUG
#   undef OUTPUTDEBUG
#   undef  LOWLEVEL
#endif

# include "cmc.h"

extern  LITERAL              *Convert_Literal_List();
extern                       Print_LITERAL_List();
extern  STRING_TABLE_ITEM    *String_Table;
extern  PREDICATE_LIST       Predicate_Table;
extern  CONSTRAINT_LIST      *Constraint_Table;
extern  int                  Number_Predicates;
extern  int                  Constraint_Number;
extern  int                  String_Table_Size;
extern  FILE                 *CM_File;
extern  FILE                 *debugf;
/*-------------------------------------------------------*/

Write_Int(out_integer,file)
    /* write out 2 byte integer to indicated file */
    int                      out_integer;
    FILE                     *file;

{
    int                      temp;

#   ifdef OUTPUTDEBUG
        fprintf(debugf,"Entering Write Int0);
#   endif
    temp = High_Byte(out_integer);
    fwrite(&temp,1,1,file);
    temp = Low_Byte(out_integer);
    fwrite(&temp,1,1,file);
}
/*-------------------------------------------------------*/

static write_string_table(string_table)
    STRING_TABLE_ITEM        *string_table;
{
    int                      temp;

    temp = High_Byte(String_Table_Size);
    fwrite(&temp,1,1,CM_File);
    temp = Low_Byte(String_Table_Size);
    fwrite(&temp,1,1,CM_File);
    while (string_table != NULL)
```

```
    {
        fwrite(string_table->String.Value,sizeof(char),
                    string_table->String.Length,CM_File);
        string_table = string_table->Next_String;
    }
}
/*-------------------------------------------------------*/

static  write_one_constraint(constraint)
    CONSTRAINT_LIST          *constraint;

/* write out entire constraint, length ,each literal */

{
    LITERAL                  *literal;
    int                      length = 0;
    int                      num_preds = 0;
    char                     output_buffer[LOCAL_BUFFER_SIZE];
    char                     *output_buffer_ptr;

#   ifdef OUTPUTDEBUG
        fprintf(debugf,"  Enter write one constraint0);
#   endif
    length += 2;
    literal = constraint->Literal_List;
#   ifdef LOWLEVEL
        fprintf(debugf,
                "Printing literal list of constraint0);
        Print_LITERAL_List(literal);
#   endif
    length += Create_Literal_List(literal,
                &(output_buffer[length]),TRUE,&num_preds);
#   ifdef OUTPUTDEBUG
        fprintf(debugf,"body length being written = %d0,
                                                length-2);
#   endif
    output_buffer[0] = High_Byte(length - 2);
    output_buffer[1] = Low_Byte(length - 2);
    fwrite(output_buffer,sizeof(char),length,CM_File);
#   ifdef LOWLEVEL
        output_buffer_ptr = output_buffer;
        output_buffer_ptr += 2;        /* to skip length */
        literal = Convert_Literal_List(&length,
                                &output_buffer_ptr,FALSE);
        fprintf(debugf,
            "Printing Converted literal constraint list0);
        Print_LITERAL_List(literal);
#   endif
}
/*-------------------------------------------------------*/

static  write_one_index_entry(predicate_index)
    PREDICATE_LIST           *predicate_index;
```

```
/* write out to CM_File one predicate name, number of
   constraints with this predicate and all constraint ids */

{
    CONTAINING_CONSTRAINTS      *constraint_list;
    int                         constraint_count = 0;
    int                         i;

#   ifdef OUTPUTDEBUG
        fprintf(debugf,"Entering write one index entry0);

        fprintf(debugf,"Predicate index name = %d0,
                                    predicate_index->Name);
#   endif
    constraint_list = predicate_index->Possible_Constraint;
    while (constraint_list != NULL)
    {
        constraint_count++;
        constraint_list = constraint_list->Next_Constraint;
    }
#   ifdef OUTPUTDEBUG
        fprintf(debugf,
            "Number of constraints, this index = %d0,
                                        constraint_count);
#   endif
    Write_Int(predicate_index->Name,CM_File);
    Write_Int(constraint_count,CM_File);
    constraint_list = predicate_index->Possible_Constraint;
    for (i = 0; i < constraint_count; i++)
    {
#       ifdef OUTPUTDEBUG
            fprintf(debugf,
                    "Constraint containing = %d0,
                            constraint_list->Constraint_Id);
#       endif
        Write_Int(constraint_list->Constraint_Id,CM_File);
        constraint_list = constraint_list->Next_Constraint;
    }
}
/*-------------------------------------------------------*/

Write_CM_File()
{
    int                         i;
    PREDICATE_LIST              *predicate_index;
    CONSTRAINT_LIST             *constraint;

#   ifdef OUTPUTDEBUG
        fprintf(debugf,"Entering Write CM File0);
        fprintf(debugf,"Number of Predicates = %d0,
                                        Number_Predicates);
        fprintf(debugf,"Next Constraint number = %d0,
                                        Constraint_Number);
```

```
#    endif
     write_string_table(String_Table);
     Write_Int(Number_Predicates,CM_File);
     predicate_index = Predicate_Table.Next_Predicate;
     for (i = 0; i < Number_Predicates; i++)
     {
         write_one_index_entry(predicate_index);
         predicate_index = predicate_index->Next_Predicate;
     }
     Write_Int(Constraint_Number - 1,CM_File);
     constraint = Constraint_Table;
     for (i = 0; i < Constraint_Number - 1; i++)
     {
         write_one_constraint(constraint);
         constraint = constraint->Next_Constraint;
     }
     fclose(CM_File);
}
/*-------------------------------------------------------------*/
```

```
/* File startup.c: This file contains the routines that are
   use to process the command line arguments.  It is a
   routine that is used by most PRISM programs.  It was
   copied and modified for the CM compiler.                */

#ifdef CMCDEBUG
#   undef STARTUPDEBUG
#endif

#include "cmc.h"

/*-------------------------------------------------------------*/
extern  TAG_TABLE           Tag_File_Table[];
extern  FILE                *Axiom_File;
extern  FILE                *Tag_File;
extern  BOOLEAN             Syntax_Only;
extern  BOOLEAN             Output_Listing;
extern  FILE                *CM_File;
extern  FILE                *debugf;
extern  int                 errno;

/*-------------------------------------------------------------*/

Get_Options(argc,argv)
    int     argc;
    char    *argv[];
{
  int    i = 1;
  int    j;
  BOOLEAN  OOPT = FALSE;
  BOOLEAN  TOPT = FALSE;

#  ifdef STARTUPDEBUG
    fprintf(debugf,"In cmc, number of args %d0,argc);
    for (i = 0; i < argc; i++)
    {
      fprintf(debugf,"Arg %d: %s0,i,argv[i]);
    }
#  endif
  i = 1;
  while (i < argc)
  {
    if (argv[i][0] == '-')
    {
      switch (argv[i][1])
      {
        case 'd':
          if ((debugf = fopen(argv[++i],"w")) == NULL)
          {
            fprintf(debugf,"Debug file '%s' ",argv[i]);
            fprintf(debugf,"could not be created0);
            exit(-1); /* to calling program */
          }
```

```
            break;
          case 'i':
            if ((Axiom_File = fopen(argv[++i],"r")) == NULL)
            {
              fprintf(debugf,"Axiom file '%s' ",argv[i]);
              fprintf(debugf,"could not be opened0);
              exit(-1); /* to calling program */
            }
            break;
          case 'l':
            Output_Listing = TRUE;
            break;
          case 'o':
            if ((CM_File = fopen(argv[++i],"w")) == NULL)
            {
              fprintf(debugf,"CM File '%s' ",argv[i]);
              fprintf(debugf,"could not be created0);
              exit(-1); /* to calling program */
            }
            OOPT = TRUE;
            break;
          case 's':
            Syntax_Only = TRUE;
            break;
          case 't':
            if ((Tag_File = fopen(argv[++i],"r")) == NULL)
            {
              fprintf(debugf,"Tag file '%s' ",argv[i]);
              fprintf(debugf,"could not be opened0);
              exit(-1); /* to calling program */
            }
            TOPT = TRUE;
            break;
          default :
            fprintf(stderr,
                  "cmc - Option %c not defined0,argv[i][1]);
        }
        i++;
      }
      else
      {
        fprintf(stderr,
          "Usage is: cmc [-d f1] -i f2 [-e f3] -o f4 -t f50);
        fprintf(stderr,"Option missing0);
        exit(-1);
      }
    }
#   ifdef STARTUPDEBUG
      fprintf(debugf,"OOPT %d TOPT %d0,OOPT,TOPT);
#   endif
    if (TOPT) then
    {
      Read_Tag_File(Tag_File,Tag_File_Table);
```

```
    fclose(Tag_File);
  }
  if (!OOPT || !TOPT)
  {
    fprintf(stderr,
            "Usage is: cmc [-d f1] -i f2 -o f3 -t f40);
    exit(-1);
  }
}
/*-------------------------------------------------------*/
```

```
/* file strings.c:  This file contains one string routine
   that was needed by the compiler that was not available
   in the PRISM library.                                    */

#ifdef CMCDEBUG
#   define  STRINGDEBUG
#endif

#include "cmc.h"

extern  STRING_TABLE_ITEM   *String_Table;
extern  BOOLEAN             Non_Null_String_Equal();
extern  FILE                *debugf;
extern  STRING_TABLE_ITEM   *Alloc_STRING_TABLE_ITEM();
extern  char                *Alloc_CHAR();
extern  int                 String_Table_Size;
/*-----------------------------------------------------------*/

Get_String_Value_Given_Offset(array,offset)

    char                    array[];
    int                     offset;
{
    STRING_TABLE_ITEM       *string_entry;

    string_entry = String_Table;
    while ((string_entry != NULL) &&
                        (string_entry->Offset != offset))
    {
        string_entry = string_entry->Next_String;
    }
    if (string_entry->Offset == offset) then
    {
        strcpy(array,string_entry->String.Value);
    }
    else
    {
        fprintf(debugf,
                "*** No string found with that offset0);
        array[0] = ' ';
    }
}
/*-----------------------------------------------------------*/
```

APPENDIX C

Appendix C contains the C code source listing of the Constraint Machine.

```
/* File cm.h: This file contains all of the data structures
   common to the files in the CM.                          */

/* # include    "prism.h" */
# include   "/ful/prism/cmc/cmc.h"

# define        MAX_NODE_ENTRIES        256
# define        MAX_QUEUE_SIZE          64

/*-------------------------------------------------------------*/
/* Predicate index structure                                   */
/*-------------------------------------------------------------*/

typedef struct
    {
        LITERAL_NAME                Name;
        CONTAINING_CONSTRAINTS      *Constraint_List;
    }
    PREDICATE_ENTRY;

/*-------------------------------------------------------------*/
/* Constraint table structure                                  */
/*-------------------------------------------------------------*/

typedef struct
    {
        LITERAL                     *Literal_List;
        BOOLEAN                     Checked_Flag;
    }
    CONSTRAINT_ENTRY;

typedef struct  partial_list_entry
    {
        LITERAL                     *Constraint;
        struct  partial_list_entry  *Next_Partial;
    }
    NODE_PARTIAL_LIST_ENTRY;

typedef struct  partial_tree_node
    {
        CLAUSE_NAME                 Node_Number;
        NODE_PARTIAL_LIST_ENTRY     *Partial;
        struct  partial_tree_node   *Parent;
        struct  partial_tree_node   *Next;
```

```
    }
    PARTIAL_TREE_NODE;

typedef struct  query_list_entry
    {
        MESSAGE_ID                      Query_Number;
        PARTIAL_TREE_NODE   *Node_Index[MAX_NODE_ENTRIES];
        PARTIAL_TREE_NODE               *Node_List;
        struct  query_list_entry        *Next_Query;
    }
    QUERY_LIST_ENTRY;

typedef struct  zmob_partial_list_entry
    {
        ZMOB_MACHINE                        Zmob;
        QUERY_LIST_ENTRY                    *Query_List;
        struct  zmob_partial_list_entry *Next_Zmob;
    }
    ZMOB_PARTIAL_LIST_ENTRY;

/*-------------------------------------------------------------*/
/* structures used in the subsumption algorithm                */
/*-------------------------------------------------------------*/

typedef enum
    {
        OKAY,
        FULL,
        PARTIAL,
        NO_VIOLATION_POSSIBLE
    }
    SUBSUMPTION_ANSWER;


typedef struct  subsumption_node
    {
        LITERAL                         *Constraint;
        LITERAL                         *Body;
        struct  subsumption_node        *Child;
        struct  subsumption_node        *Sib;
    }
    SUBSUMPTION_NODE;

typedef enum
    {
        CREATING_CHILD,
        CREATING_SIB
    }
    SUBSUMPTION_STATUS;

typedef struct  subsumption_partial_list
    {
        SUBSUMPTION_NODE                        *Answer;
```

```
        struct  subsumption_partial_list    *Next;
    }
    SUBSUMPTION_PARTIAL_LIST;
/*-------------------------------------------------------------*/
```

```
/* File alloc.c contains allocation/deallocation routines
    for structure types local to the CM.
    Alloc_<type_name> allocates a space for a structure of
    type type_name.
    Similarly for Dealloc_<type_name>                        */

#ifdef CMDEBUG
#   undef   ALLOCDEBUG
#endif

# include   "cm.h"

/*-----------------------------------------------------------*/

# define    CONSTRAINT_ENTRY_SIZE
                ((sizeof(CONSTRAINT_ENTRY)+
                        (sizeof(int)-1))/sizeof(int))

# define    PREDICATE_ENTRY_SIZE
                ((sizeof(PREDICATE_ENTRY)+
                        (sizeof(int)-1))/sizeof(int))

# define    CONTAINING_CONSTRAINTS_SIZE
                ((sizeof(CONTAINING_CONSTRAINTS)+
                        (sizeof(int)-1))/sizeof(int))

# define    NODE_PARTIAL_LIST_ENTRY_SIZE
                ((sizeof(NODE_PARTIAL_LIST_ENTRY)+
                        (sizeof(int)-1))/sizeof(int))

# define    PARTIAL_TREE_NODE_SIZE
                ((sizeof(PARTIAL_TREE_NODE)+
                        (sizeof(int)-1))/sizeof(int))

# define    QUERY_LIST_ENTRY_SIZE
                ((sizeof(QUERY_LIST_ENTRY)+
                        (sizeof(int)-1))/sizeof(int))

# define    ZMOB_PARTIAL_LIST_ENTRY_SIZE
                ((sizeof(ZMOB_PARTIAL_LIST_ENTRY)+
                        (sizeof(int)-1))/sizeof(int))

# define    SUBSUMPTION_NODE_SIZE
                ((sizeof(SUBSUMPTION_NODE)+
                        (sizeof(int)-1))/sizeof(int))

# define    SUBSUMPTION_PARTIAL_LIST_SIZE
                ((sizeof(SUBSUMPTION_PARTIAL_LIST)+
                        (sizeof(int)-1))/sizeof(int))

/*-----------------------------------------------------------*/

extern  BOOLEAN                     Using_Belt;
```

```
extern  FILE                            *debugf;
extern  char                            *alloc_block();
extern                                  Dealloc_LITERAL_List();
extern  ZMOB_PARTIAL_LIST_ENTRY         Head_Zmob_Partial_List;

/*-------------------------------------------------------*/

char *Local_Alloc(size)
    unsigned int size;
{
    char        *ptr;

/*  if (Using_Belt) then
    {
        Disable_Interrupts();
    }                               removed due to 4.2 */
    ptr = alloc_block(&size);
/*  if (Using_Belt) then
    {
        Enable_Interrupts();
    }                               removed due to 4.2 */
    return(ptr);
}

/*-------------------------------------------------------*/

PREDICATE_ENTRY *Alloc_PREDICATE_ENTRY(n)
    int     n;
{
    return((PREDICATE_ENTRY *)
                    Local_Alloc(n*(PREDICATE_ENTRY_SIZE)));
}

/*-------------------------------------------------------*/

CONSTRAINT_ENTRY *Alloc_CONSTRAINT_ENTRY(n)
    int n;
{
    return((CONSTRAINT_ENTRY *)
                Local_Alloc(n*(CONSTRAINT_ENTRY_SIZE)));
}

/*-------------------------------------------------------*/

CONTAINING_CONSTRAINTS  *Alloc_CONTAINING_CONSTRAINTS()
{
    return((CONTAINING_CONSTRAINTS *)
                Local_Alloc(CONTAINING_CONSTRAINTS_SIZE));
}
/*-------------------------------------------------------*/

NODE_PARTIAL_LIST_ENTRY *Alloc_NODE_PARTIAL_LIST_ENTRY()
{
```

```
    return((NODE_PARTIAL_LIST_ENTRY *)
                Local_Alloc(NODE_PARTIAL_LIST_ENTRY_SIZE));
}
/*-------------------------------------------------------*/

PARTIAL_TREE_NODE   *Alloc_PARTIAL_TREE_NODE()
{
    return((PARTIAL_TREE_NODE *)
                    Local_Alloc(PARTIAL_TREE_NODE_SIZE));
}
/*-------------------------------------------------------*/

QUERY_LIST_ENTRY    *Alloc_QUERY_LIST_ENTRY()
{
    return((QUERY_LIST_ENTRY *)
                    Local_Alloc(QUERY_LIST_ENTRY_SIZE));
}
/*-------------------------------------------------------*/

ZMOB_PARTIAL_LIST_ENTRY *Alloc_ZMOB_PARTIAL_LIST_ENTRY()
{
  return((ZMOB_PARTIAL_LIST_ENTRY *)
                Local_Alloc(ZMOB_PARTIAL_LIST_ENTRY_SIZE));
}
/*-------------------------------------------------------*/

SUBSUMPTION_NODE    *Alloc_SUBSUMPTION_NODE()
{
    return((SUBSUMPTION_NODE *)
                    Local_Alloc(SUBSUMPTION_NODE_SIZE));
}
/*-------------------------------------------------------*/

SUBSUMPTION_PARTIAL_LIST  *Alloc_SUBSUMPTION_PARTIAL_LIST()
{
  return((SUBSUMPTION_PARTIAL_LIST *)
                Local_Alloc(SUBSUMPTION_PARTIAL_LIST_SIZE));
}
/*-------------------------------------------------------*/

Local_Dealloc(ptr,size)
    char            *ptr;
    unsigned int    size;
{
/*  if (Using_Belt) then
    {
        Disable_Interrupts();
    }                               removed due to 4.2 */
    dealloc_block(ptr,size);
/*  if (Using_Belt) then
    {
        Enable_Interrupts();
    }                               removed due to 4.2 */
```

```
}

/*-------------------------------------------------------*/

Dealloc_CONTAINING_CONSTRAINTS_List(list)
    CONTAINING_CONSTRAINTS              *list;
{
    CONTAINING_CONSTRAINTS              *next;

#   ifdef ALLOCDEBUG
        fprintf(debugf,
            "Entering Dealloc_Containing_Constraints0);
#   endif
    while (list != NULL)
    {
        next = list->Next_Constraint;
        Local_Dealloc(list,CONTAINING_CONSTRAINTS_SIZE);
        list = next;
    }
}
/*-------------------------------------------------------*/

Dealloc_SUBSUMPTION_NODE(node)

    SUBSUMPTION_NODE                    *node;
{
#   ifdef ALLOCDEBUG
        fprintf(debugf,
                    "Enter Dealloc_Subsumption_Node0);
#   endif
    Dealloc_LITERAL_List(node->Body);
    if (node->Constraint != NULL) then
    {
        Dealloc_LITERAL_List(node->Constraint);
    }
    Local_Dealloc(node,SUBSUMPTION_NODE_SIZE);
}
/*-------------------------------------------------------*/

Dealloc_Subsumption_Tree(node_ptr)

    SUBSUMPTION_NODE                    *node_ptr;
{
#   ifdef ALLOCDEBUG
        fprintf(debugf,"Enter Dealloc_Subsumption_Tree0);
#   endif
    if (node_ptr != NULL) then
    {
        Dealloc_Subsumption_Tree(node_ptr->Child);
        Dealloc_Subsumption_Tree(node_ptr->Sib);
        Dealloc_SUBSUMPTION_NODE(node_ptr);
    }
}
```

```
/*-------------------------------------------------------------*/

Dealloc_SUBSUMPTION_PARTIAL_LIST_List(list)

    SUBSUMPTION_PARTIAL_LIST            *list;
{
    SUBSUMPTION_PARTIAL_LIST            *next;

#   ifdef ALLOCDEBUG
        fprintf(debugf
           ,"Enter Dealloc_Subsumption_Partial_List0);
#   endif
    while (list != NULL)
    {
        next = list->Next;
        Local_Dealloc(list,SUBSUMPTION_PARTIAL_LIST_SIZE);
        list = next;
    }
}
/*-------------------------------------------------------------*/

Dealloc_NODE_PARTIAL_LIST_ENTRY_List(list)

    NODE_PARTIAL_LIST_ENTRY             *list;
{
    NODE_PARTIAL_LIST_ENTRY             *next;

#   ifdef ALLOCDEBUG
        fprintf(debugf,
            "Enter Dealloc_Node_Partial_List_Entry0);
#   endif
    while (list != NULL)
    {
        next = list->Next_Partial;
        Dealloc_LITERAL_List(list->Constraint);
        Local_Dealloc(list,NODE_PARTIAL_LIST_ENTRY_SIZE);
        list = next;
    }
}
/*-------------------------------------------------------------*/

Dealloc_PARTIAL_TREE_NODE(node)

    PARTIAL_TREE_NODE                           *node;

{
#   ifdef ALLOCDEBUG
        fprintf(debugf,
                    "Enter Dealloc_Partial_Tree_Node0);
#   endif
    if (node->Partial != NULL) then
    {
        Dealloc_NODE_PARTIAL_LIST_ENTRY_List(node->Partial);
```

```
    }
    Local_Dealloc(node,PARTIAL_TREE_NODE_SIZE);
}

/*-------------------------------------------------------*/

Dealloc_PARTIAL_TREE_NODE_List(node_list)

    PARTIAL_TREE_NODE                       *node_list;
{
    PARTIAL_TREE_NODE                       *next;

#   ifdef ALLOCDEBUG
        fprintf(debugf,
            "Enter Dealloc_Partial_Tree_Node_List0);
#   endif
    while (node_list != NULL)
    {
        next = node_list->Next;
        Dealloc_PARTIAL_TREE_NODE(node_list);
        node_list = next;
    }
}
/*-------------------------------------------------------*/

Dealloc_QUERY_LIST_ENTRY(query)

    QUERY_LIST_ENTRY                        *query;
{
#   ifdef ALLOCDEBUG
        fprintf(debugf,
                    "Enter Dealloc_Query_List_Entry0);
#   endif
    Dealloc_PARTIAL_TREE_NODE_List(query->Node_List);
    Local_Dealloc(query,QUERY_LIST_ENTRY_SIZE);
}

/*-------------------------------------------------------*/

Dealloc_QUERY_LIST_ENTRY_List(list)

    QUERY_LIST_ENTRY                        *list;
{
    QUERY_LIST_ENTRY                        *next;

#   ifdef ALLOCDEBUG
        fprintf(debugf,
            "Enter Dealloc_Query_List_Entry_List0);
#   endif
    while (list != NULL)
    {
        next = list->Next_Query;
        Dealloc_QUERY_LIST_ENTRY(list);
```

```
        list = next;
    }
}
/*-------------------------------------------------------------*/

Dealloc_ZMOB_PARTIAL_LIST_ENTRY(entry)

    ZMOB_PARTIAL_LIST_ENTRY                    *entry;

{
#   ifdef ALLOCDEBUG
        fprintf(debugf,
            "Enter Dealloc_Zmob_Partial_List_Entry0);
#   endif
    Dealloc_QUERY_LIST_ENTRY_List(entry->Query_List);
    Local_Dealloc(entry,ZMOB_PARTIAL_LIST_ENTRY_SIZE);
}
/*-------------------------------------------------------------*/

Dealloc_ZMOB_PARTIAL_LIST_ENTRY_List()
    /* uses global headnode */
{
    ZMOB_PARTIAL_LIST_ENTRY                    *next,
                                               *list;

#   ifdef ALLOCDEBUG
        fprintf(debugf,
        "Enter Dealloc_Zmob_Partial_List_Entry_List0);
#   endif
    list = Head_Zmob_Partial_List.Next_Zmob;
    while (list != NULL)
    {
        next = list->Next_Zmob;
        Dealloc_ZMOB_PARTIAL_LIST_ENTRY(list);
        list = next;
    }
}
/*-------------------------------------------------------------*/
```

```
/* File cm.c: This file contains the main routine for the
   CM.  It just loops until a message arrives or the
   terminate flag is set.                                    */

# include "cm.h"

extern  FILE                    *debugf;
extern  BOOLEAN                 Using_Belt;
extern  FILE                    *Input_File;
extern  BOOLEAN                 Terminate_Flag;
extern                          Process_Msg();
extern  ZMOB_MESSAGE            *Read_PRISM_Message();

/*-----------------------------------------------------------*/

main(argc,argv)
    int     argc;
    char    *argv[];
{
    ZMOB_MESSAGE    *z_msg;

    fprintf(debugf,"CM program starting up.0);
    CM_Startup(argc,argv);
    while (!Terminate_Flag)
    {
        if (Using_Belt) then
        {
            pause();
        }
        else
        {
            z_msg = Read_PRISM_Message(Input_File);
            Process_Msg(z_msg);
        }
    }
    fprintf(debugf,
            "CM program, terminate flag set, normal stop0);
}
/*-----------------------------------------------------------*/
```

```
/* File checkconst.c:  This file contains the code for the
   routine that processes the CHECKCONST message from the
   PSM.                                                    */
#ifdef  CMDEBUG
#   define CHECKCONSTDEBUG
#   define LOWLEVEL
#endif

/*---------------------------------------------------------*/
#include "cm.h"

#   ifdef LOWLEVEL
extern                              Dump_Partial_Tree();
#   endif
extern  FILE                        *debugf;
extern
                Remove_Partial_From_Nodes_Partial_List();
extern
                    Add_Partial_Constraint_To_Node();
extern  BOOLEAN                     Find_Parent_In_Tree();
extern  PARTIAL_TREE_NODE           *Create_New_Partial_Tree();
extern
                    Replace_Partial_By_Partial_List();
extern  SUBSUMPTION_ANSWER
                    Evaluate_Nodes_Partial_List();
extern
                    Apply_Sub_List_To_Partial_List();
extern  PARTIAL_TREE_NODE           *Create_Child_Of_Parent();
extern  NODE_PARTIAL_LIST_ENTRY
                    *Copy_Partial_Constraint_List();
extern  CONTAINING_CONSTRAINTS
                    *Find_All_Constraints_To_Check();
extern  MESSAGE                     *Alloc_MESSAGE();
extern                              Output_Message();
extern                              Clear_Checked_Flags();
extern                              Dealloc_MESSAGE();
extern
                        Remove_Node_From_Partial_Tree();
extern  CONSTRAINT_ENTRY            *Constraint_Table;
extern  NODE_PARTIAL_LIST_ENTRY     *Subsumption();


/*---------------------------------------------------------*/
/*These are global variables for all routines in this file*/

BOOLEAN                             Parent_Found;
BOOLEAN                             New_Node_Created;
ZMOB_PARTIAL_LIST_ENTRY             *Zmob;
QUERY_LIST_ENTRY                    *Query_Node;
PARTIAL_TREE_NODE                   *Parent_Node;
PARTIAL_TREE_NODE                   *New_Node;
SUBSUMPTION_ANSWER                  New_Node_Status;
```

```
/*-------------------------------------------------------------*/

static      check_subsumption_old_partials_new_body(body)

/* uses global New_Node, New_Node_Status, */

    LITERAL                         *body;

{
    NODE_PARTIAL_LIST_ENTRY         *temp,
                                    *new_partial,
                                    *one_partial;
    SUBSUMPTION_ANSWER              node_status = OKAY;

#   ifdef CHECKCONSTDEBUG
        fprintf(debugf,
        "Enter check_subsump_old_partials_new_body0);
#   endif
    one_partial = New_Node->Partial;
    while (one_partial != NULL)
    {
        new_partial =
                Subsumption(body,one_partial->Constraint,
                                    &node_status);
        switch(node_status)
        {
            case NO_VIOLATION_POSSIBLE:
                temp = one_partial;
                one_partial = one_partial->Next_Partial;
                Remove_Partial_From_Nodes_Partial_List(temp,
                                                New_Node);
                break;
            case OKAY:
                one_partial = one_partial->Next_Partial;
                break;
            case FULL:
                New_Node_Status = FULL;
                return;
                break;
            case PARTIAL:
                New_Node_Status = PARTIAL;
                temp = one_partial;
                one_partial = one_partial->Next_Partial;
                Replace_Partial_By_Partial_List(temp,
                                    new_partial,New_Node);
                break;
            default:
                fprintf(debugf,
                        "Unknown new node status0);
        }
    }
}
/*-------------------------------------------------------------*/
```

```
static  check_for_new_subsumptions(body,sender,query_number,
                                   new_node_num,constraint_list)

    LITERAL                         *body;
    ZMOB_MACHINE                    sender;
    MESSAGE_ID                      query_number;
    CLAUSE_NAME                     new_node_num;
    CONTAINING_CONSTRAINTS          *constraint_list;

{

    CONTAINING_CONSTRAINTS          *list;
    NODE_PARTIAL_LIST_ENTRY         *new_partial_entry;
    SUBSUMPTION_ANSWER              node_status = OKAY;
    BOOLEAN                         already_checked = FALSE;

#   ifdef CHECKCONSTDEBUG
        fprintf(debugf,
                "Enter check_for_new_subsumptions0);
        if (constraint_list == NULL) then
        {
            fprintf(debugf,
                "ERROR constraint list = NULL0);
        }
#   endif
    for (list = constraint_list; list != NULL;
                                   list = list->Next_Constraint)
    {
    already_checked =
        Constraint_Table[list->Constraint_Id-1].Checked_Flag;
    if (!already_checked) then
    {
#   ifdef CHECKCONSTDEBUG
        fprintf(debugf,"Call subsumption on constraint %d 0,
                                            list->Constraint_Id);
#   endif
        new_partial_entry = Subsumption(body,
                Constraint_Table
                    [list->Constraint_Id - 1].Literal_List,
                        &node_status);
        switch (node_status)
        {
            case PARTIAL:
                New_Node_Status = PARTIAL;
                if (Parent_Found || New_Node_Created) then
                {
                    Add_Partial_Constraint_To_Node(New_Node,
                                            new_partial_entry);
                }
                else
                {
                    New_Node = Create_New_Partial_Tree(sender,
                                                query_number,
```

```
                                                    new_node_num,
                                                    &Zmob,
                                                    &Query_Node);
                        New_Node_Created = TRUE;
                        Add_Partial_Constraint_To_Node(New_Node,
                                                    new_partial_entry);
                    }
                    break;
                case OKAY:
                    break;
                case FULL:
                    New_Node_Status = FULL;
                    return;
                    break;
                default:
                    fprintf(debugf,
                    "Unknown status in check for new subsump0);
                    break;
            }
/*-------------------------------------------------------------*/
        }
        Constraint_Table[list->Constraint_Id-1].Checked_Flag =
                                                            TRUE;
        }
#       ifdef CHECKCONSTDEBUG
            fprintf(debugf,
                "Exit check_for_new_subsumptions---no full0);
#       endif
}
/*-------------------------------------------------------------*/

static PARTIAL_TREE_NODE *copy_and_check_parent_constraints
                                            (new_node_num,
                                            substitution)

/* only called if parent exists and global pointers set */

    CLAUSE_NAME                 new_node_num;
    SUBSTITUTION                *substitution;

{
    PARTIAL_TREE_NODE           *child;

#   ifdef CHECKCONSTDEBUG
        fprintf(debugf,
            "Enter copy_and_check_parent_constraints0);
#   endif
    child = Create_Child_Of_Parent(Parent_Node,new_node_num,
                                                    Query_Node);
    child->Partial =
        Copy_Partial_Constraint_List(Parent_Node->Partial);
    if (substitution != NULL) then
    {
```

```
        Apply_Sub_List_To_Partial_List(child->Partial,
                                                substitution);
        New_Node_Status = Evaluate_Nodes_Partial_List(child);
    }
    return(child);
}
/*-------------------------------------------------------*/

Checkconst(psm_msg)

    MESSAGE                         *psm_msg;

{

    ZMOB_MACHINE                    sender;
    MESSAGE_ID                      control_number;
    LITERAL                         *body;
    CLAUSE_NAME                     parent_num;
    CLAUSE_NAME                     new_node_num;
    SUBSTITUTION                    *sub_list;
    CONTAINING_CONSTRAINTS          *constraint_list = NULL;
    MESSAGE                         *out_msg;


#   ifdef CHECKCONSTDEBUG
        fprintf(debugf,"Enter Checkconst routine0);
#   endif
    Zmob = NULL;
    Query_Node = NULL;
    Parent_Node = NULL;
    New_Node = NULL;
    New_Node_Created = FALSE;
    New_Node_Status = OKAY;
    sender = psm_msg->Message.Checkconst.Sender;
    body = psm_msg->Message.Checkconst.Idb_Data.Body;
    control_number =
            psm_msg->Message.Checkconst.Control_Number;
    parent_num =
        psm_msg->Message.Checkconst.Connection_Info.
                                            Parent.Clause;
    sub_list = psm_msg->Message.Checkconst.Idb_Data.Sub_List;
    new_node_num =
        psm_msg->Message.Checkconst.Connection_Info.Node.
                                                Clause;
#   ifdef CHECKCONSTDEBUG
        fprintf(debugf,
                "Sender = %d, Query = %d, ",sender,
                                            control_number);
        fprintf(debugf,
                "Parent = %d, New node = %d0,
                                    parent_num,new_node_num);
#   endif
    Parent_Found = Find_Parent_In_Tree(sender,control_number,
```

```
                                                parent_num,
                                                &Zmob,&Query_Node,
                                                &Parent_Node);

        if (Parent_Found) then
        {
            /* now apply sub to old partial constraints */
            New_Node =
                copy_and_check_parent_constraints(new_node_num,
                                                  sub_list);
        }

        if ((Parent_Found) && (New_Node_Status != FULL) &&
                (New_Node_Status != NO_VIOLATION_POSSIBLE) &&
                                            (body != NULL)) then
        {
            check_subsumption_old_partials_new_body(body);
        }

            /* now check original constraints with new body */
        if ((New_Node_Status != FULL) && (body != NULL)) then
        {
            constraint_list = Find_All_Constraints_To_Check(body);
            if (constraint_list != NULL) then
            {
                check_for_new_subsumptions(body,sender,
                                           control_number,
                                           new_node_num,
                                           constraint_list);
            }
        }
        if (New_Node_Status == FULL) then
        {
            out_msg = Alloc_MESSAGE();
            out_msg->Type = VIOLATION;
            out_msg->Message.Violation.Sender = sender;
            out_msg->Message.Violation.Fail_Node.Machine =
                                                    sender;
            out_msg->Message.Violation.Control_Number =
                                           control_number;
            out_msg->Message.Violation.Fail_Node.Clause =
                                                 new_node_num;
            Output_Message(out_msg);
#           ifdef CHECKCONSTDEBUG
                fprintf(debugf,
                    "CM sent VIOLATION msg, zmob %d, query %d,",
                                        sender,control_number);
                fprintf(debugf," node %d0,new_node_num);
#           endif
            Dealloc_MESSAGE(out_msg);
        }
        if (constraint_list != NULL) then
        {
```

```
        Clear_Checked_Flags(constraint_list);
        Dealloc_CONTAINING_CONSTRAINTS_List(constraint_list);
    }
    if ((Parent_Found || New_Node_Created) &&
        ((New_Node->Partial == NULL) ||
                        (New_Node_Status == FULL))) then
    {
        Remove_Node_From_Partial_Tree(New_Node,Query_Node);
    }
#   ifdef CHECKCONSTDEBUG
        fprintf(debugf,"Exiting Checkconst...0);
#       ifdef LOWLEVEL
            Dump_Partial_Tree();
#       endif
#   endif
}
/*-------------------------------------------------------------*/
```

```
/* File cmlib.c: This file contains routines that are common
   to many procedures in the CM.  This file should be added
   to the PRISM library.                                      */

#ifdef CMDEBUG
#    undef   CMLIBDEBUG
#    undef  LOWLEVEL
#endif

#include "cm.h"

extern  FILE                            *debugf;
extern  LITERAL                         *Copy_Literal();
extern  LITERAL                         *Copy_Literal_List();
extern  NODE_PARTIAL_LIST_ENTRY
                        *Alloc_NODE_PARTIAL_LIST_ENTRY();
extern                                  Apply_Sub_List();
extern                                  Dealloc_LITERAL();
extern  CONSTRAINT_ENTRY                *Constraint_Table;
#ifdef LOWLEVEL
    extern                              Print_LITERAL_List();
#endif

/*-------------------------------------------------------------*/
Clear_Checked_Flags(list)

    CONTAINING_CONSTRAINTS          *list;
{
#   ifdef LOWLEVEL
        fprintf(debugf,"Entering Clear Flags0);
#   endif
    while (list != NULL)
    {
        Constraint_Table[list->Constraint_Id-1].Checked_Flag
                                                    = FALSE;
    list = list->Next_Constraint;
    }
}
/*-------------------------------------------------------------*/

Print_Partial_List(partial_list)

    NODE_PARTIAL_LIST_ENTRY             *partial_list;
{
    fprintf(debugf
            ,"Printing list of partial constraints0);
    while (partial_list != NULL)
    {
        Print_LITERAL_List(partial_list->Constraint);
        fprintf(debugf,"0);
        partial_list = partial_list->Next_Partial;
    }
}
```

```
/*-------------------------------------------------------*/

LITERAL *Copy_Literal_List_And_Remove_Literal(literal_list,
                                              literal_to_delete)

    LITERAL     *literal_list;
    LITERAL     *literal_to_delete;
{
    LITERAL     *first_literal,head_node;
#   ifdef CMLIBDEBUG
        fprintf(debugf,
            "Enter Copy Lit list and remove literal0);
#   endif
    head_node.Next_Literal = NULL;
    first_literal = &head_node;
    while (literal_list != NULL)
    {
        if (literal_list != literal_to_delete) then
        {
            first_literal->Next_Literal =
                            Copy_Literal(literal_list);
            first_literal = first_literal->Next_Literal;
        }
        literal_list = literal_list->Next_Literal;
    }
    first_literal->Next_Literal = NULL;
#   ifdef CMLIBDEBUG
        fprintf(debugf,
            "Exit Copy LITERAL list..literal list is:0);
#       ifdef LOWLEVEL
            Print_LITERAL_List(head_node.Next_Literal);
            fprintf(debugf,"0);
#       endif
#   endif
    return(head_node.Next_Literal);
}
/*-------------------------------------------------------*/


NODE_PARTIAL_LIST_ENTRY  *Copy_Partial_Constraint_List(list)

    NODE_PARTIAL_LIST_ENTRY             *list;

{
    NODE_PARTIAL_LIST_ENTRY             headnode, *temp;

#   ifdef CMLIBDEBUG
        fprintf(debugf,"In Copy_Partial_Constraint_List0);
#   endif
    headnode.Next_Partial = NULL;
    temp = &headnode;
    while (list != NULL)
    {
```

```
        temp->Next_Partial = Alloc_NODE_PARTIAL_LIST_ENTRY();
        temp = temp->Next_Partial;
        temp->Next_Partial = NULL;
        temp->Constraint =
                        Copy_Literal_List(list->Constraint);
        list = list->Next_Partial;
    }
#   ifdef CMLIBDEBUG
#       ifdef LOWLEVEL
            Print_Partial_List(headnode.Next_Partial);
#       endif
#   endif
    return(headnode.Next_Partial);
}
/*-------------------------------------------------------*/


Apply_Sub_List_To_Partial_List(partial_list,substitution)

    NODE_PARTIAL_LIST_ENTRY             *partial_list;
    SUBSTITUTION                        *substitution;

{
    NODE_PARTIAL_LIST_ENTRY             *temp;

#   ifdef CMLIBDEBUG
        fprintf(debugf,
                "Enter Apply_Sub_List_To_Partial_List0);
#       ifdef LOWLEVEL
            Print_Partial_List(partial_list);
#       endif
#   endif
    temp = partial_list;
    while (partial_list != NULL)
    {
        Apply_Sub_List(partial_list->Constraint,
                                            substitution);
        partial_list = partial_list->Next_Partial;
    }
#   ifdef LOWLEVEL
        Print_Partial_List(temp);
#   endif
}
/*-------------------------------------------------------*/

BOOLEAN Fully_Instantiated(literal)

    LITERAL                         *literal;
{
    TERM                            *term;

    for (term = literal->Arg_List; term != NULL;
                                        term = term->Next_Term)
```

```
    {
        if ((term->Type == CM_VARIABLE) ||
                            (term->Type == IDB_VARIABLE) ||
                            (term->Type == VARIABLE)) then
        {
            return(FALSE);
        }
    }
    return(TRUE);
}
/*-------------------------------------------------------------*/
```

```
/*  This file contains all commands that the Constraint
    Machine processes except for the Checkconst command */

#ifdef  CMDEBUG
#   undef   COMMANDSDEBUG
#   undef   ALLOCDEBUG
#endif

#ifdef VAXDEBUG
extern              Dump_Constraint_Table();
extern              Dump_Predicate_Table();
extern              Dump_Partial_Tree();
#endif

# include "cm.h"

extern  FILE        *debugf;
extern  BOOLEAN     Terminate_Flag;
extern  BOOLEAN     Using_Belt;
extern  MESSAGE     *Alloc_MESSAGE();
extern              *Erase_Zmob_Query_List_Entry();
extern  BOOLEAN     Find_Zmob();
extern  BOOLEAN     Find_Query_Node();

/*-------------------------------------------------------*/

Erase(msg)
/* Proceses ERASE: Deletes query list corresp to the source
                   and control number */

    MESSAGE                         *msg;
{
    ZMOB_MACHINE                    source_machine;
    MESSAGE_ID                      control_number;
    MESSAGE                         *msgout;
    BOOLEAN                         found;
    ZMOB_PARTIAL_LIST_ENTRY         *zmob_ptr;
    QUERY_LIST_ENTRY                *query_node_ptr;


#   ifdef COMMANDSDEBUG
        fprintf(debugf,"Entering Erase procedure0);
#   endif
    source_machine = msg->Message.Erase.Sender;
    control_number = msg->Message.Erase.Control_Number;
#   ifdef   COMMANDSDEBUG
        fprintf(debugf,"CM erase: source= %d , control=%d0,
                source_machine,control_number);
#   endif
    found = Find_Zmob(source_machine,&zmob_ptr);
    if (found) then
    {
#       ifdef COMMANDSDEBUG
```

```
            fprintf(debugf,
                    "Source zmob found in partial lists0);
#       endif
        found = Find_Query_Node(control_number,
                                &query_node_ptr,
                                zmob_ptr);
        if (found) then
        {
#           ifdef COMMANDSDEBUG
                fprintf(debugf,
                        "Query found in partial tree0);
#           endif
            Erase_Zmob_Query_List_Entry(zmob_ptr,
                                        query_node_ptr);
        }
        else
        {
#           ifdef COMMANDSDEBUG
                fprintf(debugf,
                    "Query NOT found in partial tree0);
#           endif
        }
    }
    else
    {
#   ifdef COMMANDSDEBUG
        fprintf(debugf,
                "Source zmob NOT found in partial lists0);
#   endif
    }
}
/*-------------------------------------------------------------*/

Terminate()
{
#   ifdef CMDEBUG
        Terminate_Flag = TRUE;
        fprintf(debugf,"Terminate routine entered0);
        return;
#   else
        Abort(FALSE);
#   endif
}
/*-------------------------------------------------------------*/

Askdump(msg)
    MESSAGE *msg;
{
#   ifdef VAXDEBUG
        Dump_Predicate_Table();
        Dump_Constraint_Table();
        Dump_Partial_Tree();
#   endif
```

```
    return;

}
/*-------------------------------------------------------------*/

Getstat(msg)
    MESSAGE *msg;
{
    return;
}
/*-------------------------------------------------------------*/

Load(msg)
    MESSAGE *msg;
{
    return;
}
/*-------------------------------------------------------------*/
```

```
/* File deallocmsg.c: This file contains the routine that
   does special processing to deallocate specific types
   of messages.                                              */
#ifdef CMDEBUG
#   undef   DEALLOCDEBUG
#   undef   LOWLEVEL
#endif

# include   "cm.h"

extern  FILE    *debugf;

/*-----------------------------------------------------------*/

Dealloc_MESSAGE(pmsg)
    MESSAGE     *pmsg;
{

#   ifdef DEALLOCDEBUG
        fprintf(debugf,
                "Deallocating MESSAGE of type%d0,
                                                pmsg->Type);
#   endif
    switch (pmsg->Type)
    {
        case ASKDUMP:
            break;
        case CHECKCONST:

            Dealloc_LITERAL_List(
                    pmsg->Message.Checkconst.Idb_Data.Body);
            Dealloc_SUBSTITUTION_List(
                pmsg->Message.Checkconst.Idb_Data.Sub_List);
            break;

        case COMDEBUG:
            break;
        case COMDEBUGOFF:
            break;
        case DEBUG:
            break;
        case DUMP:
            break;
        case ERASE:
            break;
        case ERASEALL:
            break;
        case ERASED:
            break;
        case ERROR:
            break;
        case GETSTATISTICS:
            break;
```

```
        case LOAD:
            break;
        case RESUME:
            break;
        case SETCOMDEBUG:
            break;
        case SETDEBUG:
            break;
        case SETSTATS:
            break;
        case STATISTICS:
            break;
        case STATSOFF:
            break;
        case SUSPEND:
            break;
        case TERMINATE:
            break;
        case VIOLATION:
            break;
        default:
            fprintf(debugf,
                    "invalid message %d in deallocator0,
                            pmsg->Type);
            break;
    }
    Local_Dealloc(pmsg,MESSAGE_SIZE);
#   ifdef DEALLOCDEBUG
        fprintf(debugf,"Deallocated MESSAGE0);
#   endif
}
/*-------------------------------------------------------------*/
```

```
/* File dump.c: Contains all routines to dump predicate
   index, constraint table, and partial constraint tree    */

#ifdef CMDEBUG
#   undef DUMPDEBUG
#endif

# include   "cm.h"

extern                                  Print_LITERAL_List();
extern                                  Print_Partial_List();
extern  int                             Find_Encoded_Name();
extern  FILE                            *debugf;
extern  ZMOB_PARTIAL_LIST_ENTRY         Head_Zmob_Partial_List;
extern  int                             Constraint_Count;
extern  int                             Number_Predicates;
extern  TAG_TABLE                       Tag_File_Table[];
extern  PREDICATE_ENTRY                 *Predicate_Table;
extern  CONSTRAINT_ENTRY                *Constraint_Table;

/*-------------------------------------------------------*/

Dump_Node_List(tree_node)

    PARTIAL_TREE_NODE                       *tree_node;
{

    fprintf(debugf,"  Dumping a Node List0);
    if (tree_node == NULL) then
    {
        fprintf(debugf,"Node list is NULL0);
    }
    else
    {
        while (tree_node != NULL)
        {
            fprintf(debugf,
                    "Node = %d",tree_node->Node_Number);
            if (tree_node->Parent != NULL) then
            {
                fprintf(debugf,"parent = %d0,
                            tree_node->Parent->Node_Number);
            }
            else
            {
                fprintf(debugf,"node is root0);
            }
            Print_Partial_List(tree_node->Partial);
            tree_node = tree_node->Next;
        }
    }
    fprintf(debugf,"  Finished dumping a Node List0);
}
```

```
/*-------------------------------------------------------------*/

Dump_Query_List(query_list)

    QUERY_LIST_ENTRY                        *query_list;
{

    fprintf(debugf,"Dumping a query list0);
    if (query_list == NULL) then
    {
        fprintf(debugf,"  Query list is NULL0);
    }
    else
    {
        while (query_list != NULL)
        {
            fprintf(debugf,"  Query = %d0,
                                query_list->Query_Number);
            Dump_Node_List(query_list->Node_List);
            query_list = query_list->Next_Query;
        }
    }
    fprintf(debugf,"Done dumping a query list0);
}
/*-------------------------------------------------------------*/

Dump_Partial_Tree()

{
    ZMOB_PARTIAL_LIST_ENTRY                 *zmob;

    fprintf(debugf,
                "0eginning to Dump entire Partial Tree0);
    zmob = Head_Zmob_Partial_List.Next_Zmob;
    if (zmob == NULL) then
    {
        fprintf(debugf,"NO zmobs in partial tree0);
    }
    else
    {
        while (zmob != NULL)
        {
            fprintf(debugf,"  Zmob = %d0,zmob->Zmob);
            Dump_Query_List(zmob->Query_List);
            zmob = zmob->Next_Zmob;
        }
    }
    fprintf(debugf,"0one dumping entire Partial Tree0);
}
/*-------------------------------------------------------------*/

Dump_Constraint_Table()
{
```

```
    int                 i;

    fprintf(debugf,
                "0Starting to dump Constraint Table0);
    for (i = 0;i < Constraint_Count; i++)
    {
    fprintf(debugf,"Constraint Number: %d0,i+1);
    Print_LITERAL_List(Constraint_Table[i]);
    fprintf(debugf,"0);
    }
    fprintf(debugf,"Finished dumping Constraint Table0);
}
/*-------------------------------------------------------*/

Dump_Predicate_Table()
{
    char                        *string_name;
    int                         i,j;
    CONTAINING_CONSTRAINTS      *constraint_list;

    fprintf(debugf,"0Dumping Predicate Table0);
    for (i = 0; i < Number_Predicates; i++)
    {
        j = Find_Encoded_Name(Predicate_Table[i].Name,
                              Tag_File_Table);
        string_name = Tag_File_Table[j].Name;
        fprintf(debugf,"
                    Predicate Number = %d,name = %s0,
                        Predicate_Table[i].Name,string_name);
        constraint_list = Predicate_Table[i].Constraint_List;
        fprintf(debugf,
                "Constraints containing predicate are:0);
        while (constraint_list != NULL)
        {
            fprintf(debugf,
                "%d0,constraint_list->Constraint_Id);
            constraint_list =
                            constraint_list->Next_Constraint;
        }
    }
}
/*-------------------------------------------------------*/
```

```
/* File evalconst.c: This file contains the routines to
   evaluate equality predicates                                    */

#ifdef  CMDEBUG
#   undef EVALCONSTDEBUG
#   undef LOWLEVEL
#endif

# define        local_EVAL_EQ                          2

#include "cm.h"

extern  FILE                            *debugf;
extern  BOOLEAN                         *Fully_Instantiated();
extern  int                             Number_Of_Arguments();

/*-------------------------------------------------------------*/

static BOOLEAN compare_terms(first_term,second_term)
    TERM                *first_term;
    TERM                *second_term;
{
    int                 i;

#   ifdef LOWLEVEL
        fprintf(debugf,"Enter compare_terms0);
#   endif
    switch(first_term->Type)
    {
        case INTEGER:
            if (second_term->Type == INTEGER) then
            {
                return(first_term->Value.Integer.Value ==
                        second_term->Value.Integer.Value);
            }
            return(FALSE);
            break;
        case STRING:
            if (second_term->Type == STRING) then
            {
            return(Non_Null_String_Equal(
                                    first_term->Value.String,
                                second_term->Value.String));
            }
            return(FALSE);
            break;
        default:
            return(FALSE);
            break;
    }
}
/*-------------------------------------------------------------*/
```

```
static  BOOLEAN     local_evaluable(literal)

    LITERAL                             *literal;

{
#   ifdef LOWLEVEL
        fprintf(debugf,"Entering local_evaluable0);
#   endif
    return((literal->Location == BUILT_IN)
            && (literal->Name == local_EVAL_EQ)
            && (Fully_Instantiated(literal))
            && (Number_Of_Arguments(literal->Arg_List) ==2));
}
/*-------------------------------------------------------*/

LITERAL     *Evaluate_One_Literal_List(literal_list,status)

    LITERAL                             *literal_list;
    SUBSUMPTION_ANSWER                  *status;
{
    BOOLEAN                             on_first_literal = TRUE;
    LITERAL                             *literal,
                                        *temp,
                                        *back = NULL;
    BOOLEAN                             answer;

#   ifdef EVALCONSTDEBUG
        fprintf(debugf,
                "Entering Evaluate_One_Literal_List0);
#       ifdef LOWLEVEL
            Print_LITERAL_List(literal_list);
            fprintf(debugf,"0);
#       endif
#   endif
    *status = OKAY;
    literal = literal_list;
    while (literal != NULL)
    {
        if (local_evaluable(literal)) then
        {
#           ifdef LOWLEVEL
                fprintf(debugf,"literal is evaluable0);
#           endif
            answer = compare_terms(literal->Arg_List,
                                   literal->Arg_List->Next_Term);
            if (answer) then
            {
#               ifdef LOWLEVEL
                    fprintf(debugf,
                            "evaluation succeeds!0);
#               endif
                if (on_first_literal) then
                {
```

```
                    literal_list = literal->Next_Literal;
                }
                else
                {
                    back->Next_Literal =
                                literal->Next_Literal;
                }
                temp = literal;
                literal = literal->Next_Literal;
                temp->Next_Literal = NULL;
                Dealloc_LITERAL(temp);
            }
            else /* fails, so cannot be violated */
            {
#               ifdef EVALCONSTDEBUG
                    fprintf(debugf," evaluation failed0);
                    fprintf(debugf,
                        " returning status NO_VIOL_POS0);
#               endif
                *status = NO_VIOLATION_POSSIBLE;
                Dealloc_LITERAL_List(literal_list);
                return(NULL);
            }
        }
        else /* just check next literal */
        {
            back = literal;
            literal = literal->Next_Literal;
            on_first_literal = FALSE;
        }
    }
    if (literal_list == NULL) then
    {
#       ifdef EVALCONSTDEBUG
            fprintf(debugf,"  returning status FULL0);
#       endif
        *status = FULL;/* if all preds deleted by eval:FULL*/
    }
    return(literal_list);
}
/*-------------------------------------------------------*/

SUBSUMPTION_ANSWER  Evaluate_Nodes_Partial_List(tree_node)

    PARTIAL_TREE_NODE               *tree_node;

{

    NODE_PARTIAL_LIST_ENTRY         *back,
                                    *partial,
                                    *temp;
    BOOLEAN                         on_first_partial = TRUE;
    LITERAL                         *modified_constraint;
```

```
    SUBSUMPTION_ANSWER                  one_constraint_status,
                                        node_status = OKAY;

#   ifdef EVALCONSTDEBUG
        fprintf(debugf,
                "  Enter Evaluate_Nodes_Partial_List0);
#   endif
    back = NULL;
    partial = tree_node->Partial;
    while (partial != NULL)
    {
        partial->Constraint =
            Evaluate_One_Literal_List(partial->Constraint,
                                      &one_constraint_status);
        switch (one_constraint_status)
        {
            case NO_VIOLATION_POSSIBLE:
                if (on_first_partial) then
                {
                    tree_node->Partial =
                                      partial->Next_Partial;
                }
                else
                {
                    back->Next_Partial =
                                      partial->Next_Partial;
                }
                temp = partial;
                partial = partial->Next_Partial;
                temp->Next_Partial = NULL;
                Dealloc_NODE_PARTIAL_LIST_ENTRY_List(temp);
                break;
            case FULL:
                node_status = FULL;
                return(node_status);
            case PARTIAL:
            case OKAY:
                back = partial;
                partial = partial->Next_Partial;
                on_first_partial = FALSE;
                break;
            default:
                fprintf(debugf,
                "Unknown status in Eval Nodes Part List0);
                break;
        }
    } /* end while */
    if (tree_node->Partial == NULL) then
    {
        node_status = NO_VIOLATION_POSSIBLE;
    }
#   ifdef EVALCONSTDEBUG
        fprintf(debugf,
```

```
                        "Exit Evaluate_Nodes_Partial_List0);
#   endif
    return(node_status);
}
/*-------------------------------------------------------------*/
```

```
/* Reads in the CM's database, sets up the string table,
   predicate index and constraint table */

#ifdef CMDEBUG
#   undef   INITSDEBUG
#endif

# include "cm.h"

extern  FILE                        *debugf;
extern  int                         Number_Predicates;
extern  int                         Constraint_Count;
extern  PREDICATE_ENTRY             *Predicate_Table;
extern  CONSTRAINT_ENTRY            *Constraint_Table;
extern  LITERAL                     *Convert_Literal_List();
extern                              Convert_IDB_Strings();
extern  char                        *Read_String_Table();
extern  PREDICATE_ENTRY             *Alloc_PREDICATE_ENTRY();
extern  CONTAINING_CONSTRAINTS
                            *Alloc_CONTAINING_CONSTRAINTS();
extern  CONSTRAINT_ENTRY            *Alloc_CONSTRAINT_ENTRY();

/*-------------------------------------------------------*/

static  set_up_cm_strings(literal_list,string_table)
    LITERAL             *literal_list;
    char                *string_table;

{
    LITERAL             *literal;

#   ifdef INITSDEBUG
        fprintf(debugf,"Setting up CM strings0);
#   endif
    for (literal = literal_list;literal != NULL;
                            literal = literal->Next_Literal)
    {
        Convert_IDB_Strings(literal->Arg_List,string_table);
    }
#   ifdef INITSDEBUG
        fprintf(debugf,"Finished setting up CM strings0);
#   endif
}
/*-------------------------------------------------------*/

static  LITERAL *read_constraint_entry(cm_file,string_table)
    FILE                *cm_file;
    char                *string_table;

{
    LITERAL             *literal_pointer;
    int                 length = 0;
    char                input_buffer[LOCAL_BUFFER_SIZE];
```

```
    char                    *temp_ptr;

#   ifdef INITSDEBUG
        fprintf(debugf,"  Enter read constraint entry0);
#   endif
    length = Read_Two_Bytes(cm_file);
    length = Read_Two_Bytes(cm_file);
#   ifdef INITSDEBUG
        fprintf(debugf,
                "Length, constraint before Convert Lit =
                                                %d,0,lenght);
#   endif
    input_buffer[0] = High_Byte(length);
    input_buffer[1] = Low_Byte(length);
    fread(&(input_buffer[2]),sizeof(char),length,cm_file);
    temp_ptr = input_buffer;
    literal_pointer = Convert_Literal_List(&length,&temp_ptr,
                                                TRUE);
    set_up_cm_strings(literal_pointer,string_table);
    return(literal_pointer);
}
/*-------------------------------------------------------*/

static read_index_entry(predicate_entry,cm_file,string_table)

/* reads in one predicate name, and constaining constraints.
   Sets up the index entry for this prediate                */

    PREDICATE_ENTRY         *predicate_entry;
    FILE                    *cm_file;
    char                    *string_table;
{
    int                     constraint_count;
    int                     i;
    CONTAINING_CONSTRAINTS      surrogate_node;
    CONTAINING_CONSTRAINTS      *constraint;

#   ifdef INITSDEBUG
        fprintf(debugf,"Reading an index entry0);
#   endif
    constraint = &surrogate_node;
    predicate_entry->Name = Read_Two_Bytes(cm_file);
    constraint_count = Read_Two_Bytes(cm_file);
    for (i = 0; i < constraint_count; i++)
    {
        constraint->Next_Constraint =
                            Alloc_CONTAINING_CONSTRAINTS();
        constraint = constraint->Next_Constraint;
        constraint->Constraint_Id = Read_Two_Bytes(cm_file);
    }
    constraint->Next_Constraint = NULL;
    predicate_entry->Constraint_List =
                            surrogate_node.Next_Constraint;
```

```
#   ifdef INITSDEBUG
        fprintf(debugf,"Finished reading one pred entry0);
#   endif
}
/*-------------------------------------------------------*/

Read_Constraint_File(cm_file)
    FILE                *cm_file;
{
    int                 i;
    char                *string_table;

#   ifdef INITSDEBUG
        fprintf(debugf,  "Enter Read Constraint File0);
#   endif
    string_table = Read_String_Table(cm_file);
    Number_Predicates = Read_Two_Bytes(cm_file);
    Predicate_Table =
                    Alloc_PREDICATE_ENTRY(Number_Predicates);
    for (i = 0; i < Number_Predicates; i++)
    {
        read_index_entry(&(Predicate_Table[i]),cm_file,
                                            string_table);
    }
#   ifdef INITSDEBUG
        fprintf(debugf,
                    "Predicate Index read from CM_File0);
        fprintf(debugf,"0eginning to read constraints0);
#   endif
    /* now to read in the constraint table */
    Constraint_Count = Read_Two_Bytes(cm_file);
#   ifdef INITSDEBUG
        fprintf(debugf,
                "Constraint Count = %d0,Constraint_Count);
#   endif
    Constraint_Table =
                    Alloc_CONSTRAINT_ENTRY(Constraint_Count);
    for (i = 0; i < Constraint_Count; i++)
    {
        Constraint_Table[i].Literal_List =
                            read_constraint_entry(cm_file,
                                            string_table);
        Constraint_Table[i].Checked_Flag = FALSE;
    }
#   ifdef INITSDEBUG
        fprintf(debugf,
                    "Finished reading constraint table0);
#   endif
}
/*-------------------------------------------------------*/
```

```
/* File input.c: Contains processes an input zmob message
   and creates a message.  Calls routine to process the
   message.                                                */

#ifdef CMDEBUG
#   define  INPUTDEBUG
#   define  ALLOCDEBUG
#endif

# include "cm.h"

extern  PORT_ID             Receive_Port;
extern  ZMOB_MACHINE        Myself;
extern  FILE                *debugf;
extern  BOOLEAN             Comdebug_Enabled;
extern  BOOLEAN             Debug_Enabled;
extern  BOOLEAN             Suspended;
extern  ZMOB_MESSAGE        *Read_PRISM_Message();
extern  MESSAGE_TYPE        Message_Type();
/* extern   ZMOB_MESSAGE    *Get_PRISM_Message(); for 4.2 */
extern  MESSAGE             *Alloc_MESSAGE();
extern  MESSAGE             *Convert_Eraseall();
extern  MESSAGE             *Convert_Erase();
extern  MESSAGE             *Convert_Askdump();
extern  MESSAGE             *Convert_Getstatistics();
extern  MESSAGE             *Convert_Load();
extern  MESSAGE             *Make_Comdebug_Message();
extern  MESSAGE             *Convert_Check_Constraint();

/*-------------------------------------------------------------*/

Process_Msg(z_msg)
    ZMOB_MESSAGE    *z_msg;
{
    MESSAGE         *msg;
    MESSAGE_ID      control_number;

#   ifdef   INPUTDEBUG
        fprintf(debugf,
                "Process Input Message, Type= %d0,
                                    Message_Type(z_msg));
#   endif
    if (z_msg==NULL) then
    {
        return;
    }
    switch(Message_Type(z_msg))
    {
        case ASKDUMP:
            msg = Convert_Askdump(z_msg);
            Askdump(msg);
            Dealloc_MESSAGE(msg);
            break;
```

```
        case CHECKCONST:
            msg = Convert_Check_Constraint(z_msg);
            msg->Message.Checkconst.Sender = Source(z_msg);
            /* should be in convert routine */
            Checkconst(msg);
            Dealloc_MESSAGE(msg);
            break;
        case COMDEBUGOFF:
            Comdebug_Enabled = FALSE;
            break;
        case DEBUGOFF:
            Debug_Enabled = FALSE;
            break;
        case ERASE:
            msg = Convert_Erase(z_msg);
            Erase(msg);
            Dealloc_MESSAGE(msg);
            break;
        case GETSTATISTICS:
            msg = Convert_Getstatistics(z_msg);
            Getstat(msg);
            Dealloc_MESSAGE(msg);
            break;
        case LOAD:
            msg = Convert_Load(z_msg);
            Load(msg);
            Dealloc_MESSAGE(msg);
            break;
        case SETCOMDEBUG:
            Comdebug_Enabled = TRUE;
            break;
        case SETDEBUG:
            Debug_Enabled = TRUE;
            break;
        case SETSTATS:
            break;
        case STATSOFF:
            break;
        case TERMINATE:
            Terminate();
            break;
        default:
            fprintf(debugf,
                    "Unknown message type %d0,
                                Message_Type(z_msg));
            break;
    }
#   ifdef INPUTDEBUG
        fprintf(debugf,"Exiting Process_Msg0);
#   endif
}
/*-------------------------------------------------------*/
```

```
/*  The following routine removed due to 4.2

Input_Interrupt_Handler()
{
    ZMOB_MESSAGE    *z_msg;

    while (ipcgetsignal(NORMALMSG) != 0)
    {
#       ifdef ALLOCDEBUG
            fprintf(debugf,"Enter Interrupt Handler Loop0);
#       endif
        z_msg = Get_PRISM_Message(Receive_Port);
        if ((z_msg == NULL) || (((int)z_msg) < 0)) then
        {
            fprintf(debugf,"CM%d Bad Input Message0,
                                                    Myself);
        }
        else
        {
            Process_Msg(z_msg);
            Dealloc_ZMOB_MESSAGE(z_msg);
        }
        ipcdismisssignal(NORMALMSG);
#       ifdef ALLOCDEBUG
            fprintf(debugf,"Exit Interrupt Handler Loop0);
#       endif
    }
}
                                                              */
/*-------------------------------------------------------------*/
```

```
/* File output.c: Contains routine to create a ZMOB message
   from a message.  Then sends it over the belt to the
   destination or writes it to a file.  The BOOLEAN
   Using_Belt determines which.                              */

# include  "cm.h"

# ifdef CMDEBUG
#   define OUTPUTDEBUG
# endif


extern  BOOLEAN          Using_Belt;
extern  BOOLEAN          Comdebug_Enabled;
extern  PORT_ID          Transmit_Port;
extern  FILE             *debugf;
extern  FILE             *Output_File;
extern  ZMOB_MACHINE     Myself;
/* extern   int      Send_PRISM_Message();removed due to 4.2*/
extern  MESSAGE          *Make_Comdebug_Message();
extern  MESSAGE          *Alloc_MESSAGE();
extern  ZMOB_MESSAGE     *Create_Comdebug();
extern  ZMOB_MESSAGE     *Create_Debug();
extern  ZMOB_MESSAGE     *Create_Dump();
extern  ZMOB_MESSAGE     *Create_Erased();
extern  ZMOB_MESSAGE     *Create_Error();
extern  ZMOB_MESSAGE     *Create_Statistics();
extern  ZMOB_MESSAGE     *Create_Violation();

/*-----------------------------------------------------------*/

ZMOB_MESSAGE             *Make_Output_Message(msg)
    MESSAGE              *msg;
{
    ZMOB_MESSAGE         *z_msg;

#   ifdef OUTPUTDEBUG
        fprintf(debugf,"Creating ZMOB message: type %d0,
                                                  msg->Type);
#   endif
    switch (msg->Type)
    {
    case COMDEBUG:
        z_msg = Create_Comdebug(msg);
        z_msg->controlword.control_register = (int)
                                             SINGLE_PATTERN;
        z_msg->controlword.destination = HOST_PATTERN;
        break;
    case DEBUG:
        z_msg = Create_Debug(msg);
        z_msg->controlword.control_register = (int)
                                             SINGLE_PATTERN;
        z_msg->controlword.destination = HOST_PATTERN;
```

```
        break;
    case ERASED:
        z_msg = Create_Erased(msg);
        z_msg->controlword.control_register = (int)
                                            SINGLE_ADDRESS;
        z_msg->controlword.destination =
                                msg->Message.Erased.Sender;
        break;
    case ERROR:
        z_msg = Create_Error(msg);
        z_msg->controlword.control_register = (int)
                                            SINGLE_ADDRESS;
        z_msg->controlword.destination =
                            msg->Message.Answer_List.Sender;
        break;
    case STATISTICS:
        z_msg = Create_Statistics(msg);
        z_msg->controlword.control_register = (int)
                                            SINGLE_PATTERN;
        z_msg->controlword.destination = HOST_PATTERN;
        break;
    case VIOLATION:
        z_msg = Create_Violation(msg);
        z_msg->controlword.control_register = (int)
                                            SINGLE_ADDRESS;
        z_msg->controlword.destination =
                            msg->Message.Violation.Sender;
        break;
    default:
        fprintf(debugf,"Unknown Output Message %d0,
                                            msg->Type);
        return(NULL);
        break;
    }
    z_msg->controlword.source = Myself;
#   ifdef OUTPUTDEBUG
        fprintf(debugf,"ZMOB message creation done0);
#   endif
    return(z_msg);
}
/*-------------------------------------------------------*/

Output_Message(out_msg)
    MESSAGE         *out_msg;
{
    MESSAGE         *comdebug_msg;
    ZMOB_MESSAGE    *out_z_msg;
    ZMOB_MESSAGE    *comdebug_z_msg;
    int             status;
    int             comdebug_status;

#   ifdef OUTPUTDEBUG
        fprintf(debugf,"Entering Output Message0);
```

```
#   endif
    out_z_msg = Make_Output_Message(out_msg);
    if (out_z_msg == NULL) then
    {
        return;
    }
    if (Using_Belt) then
    {
#       ifdef OUTPUTDEBUG
            fprintf(debugf,"Sending Output Message0);
#       endif
/*      Disable_Interrupts();           removed due to 4.2
        status = Send_PRISM_Message(out_z_msg,Transmit_Port);

        Enable_Interrupts();                    for 4.2 */
#       ifdef OUTPUTDEBUG
            fprintf(debugf,
                "Sent Output Message to port: status %d0,
                                                    status);
#       endif
/*--------------------------------------------------------*/
/*      if (status == 0)then
        {
            fprintf(debugf
                        "CM%d send to port %d timed out0,
                                    Myself,Transmit_Port);
        }
        else if (status == -1)then
        {
            Abort(TRUE);
        }
*/
    }
    else
    {
        Write_PRISM_Message(Output_File,out_z_msg);
    }
    Dealloc_ZMOB_MESSAGE(out_z_msg);
#   ifdef OUTPUTDEBUG
        fprintf(debugf,"Sent Output Message0);
#   endif
}
/*--------------------------------------------------------*/
```

```
/* File partials.c: Contains all routines to manipulate the
   tree of partial constraints.                              */

#ifdef  CMDEBUG
#   undef PARTIALSDEBUG
#   undef LOWLEVEL
#endif

#include "cm.h"

extern  FILE                        *debugf;
extern  ZMOB_PARTIAL_LIST_ENTRY Head_Zmob_Partial_List;
extern  ZMOB_PARTIAL_LIST_ENTRY
                            *Alloc_ZMOB_PARTIAL_LIST_ENTRY();
extern  QUERY_LIST_ENTRY            *Alloc_QUERY_LIST_ENTRY();
extern  PARTIAL_TREE_NODE           *Alloc_PARTIAL_TREE_NODE();
extern                              Dealloc_PARTIAL_TREE_NODE();
#ifdef LOWLEVEL
    extern                          Print_LITERAL_List();
#endif

/*-----------------------------------------------------------*/

Remove_Partial_From_Nodes_Partial_List(partial,node)

    NODE_PARTIAL_LIST_ENTRY             *partial;
    PARTIAL_TREE_NODE                   *node;
{
    NODE_PARTIAL_LIST_ENTRY             *back = NULL,
                                        *temp;

#   ifdef PARTIALSDEBUG
        fprintf(debugf,
        "Enter Remove_Partial_From_Nodes_Partial_List0);
#   endif
    if (node->Partial == partial) then
    {
        node->Partial = partial->Next_Partial;
    }
    else
    {
        temp = node->Partial;
        while ((temp != NULL) && (temp != partial))
        {
            back = temp;
            temp = temp->Next_Partial;
        }
        back->Next_Partial = partial->Next_Partial;
        partial->Next_Partial = NULL;
    }
    Dealloc_NODE_PARTIAL_LIST_ENTRY_List(partial);
}
/*-----------------------------------------------------------*/
```

```
static  insert_node_in_index(node_num,query_node,node_ptr)

    CLAUSE_NAME                 node_num;
    QUERY_LIST_ENTRY            *query_node;
    PARTIAL_TREE_NODE           *node_ptr;
{
    query_node->Node_Index[node_num % MAX_NODE_ENTRIES] =
                                                    node_ptr;
}
/*-----------------------------------------------------------*/

static  clear_index(query_node)

    QUERY_LIST_ENTRY            *query_node;

{
    int                         i;

    for (i = 0; i < MAX_NODE_ENTRIES;
                        query_node->Node_Index[i++] = NULL);
}
/*-----------------------------------------------------------*/

static PARTIAL_TREE_NODE *create_partial_tree_node(
                                                query_node_ptr,
                                                    new_node_num)

    QUERY_LIST_ENTRY            **query_node_ptr;
    CLAUSE_NAME                 new_node_num;
{

    PARTIAL_TREE_NODE           *temp;

#   ifdef LOWLEVEL
        fprintf(debugf,"Enter create_partial_tree_node0);
#   endif
    temp = Alloc_PARTIAL_TREE_NODE();
    temp->Partial = NULL;
    temp->Parent = NULL; /*    10 Aug */
    temp->Node_Number = new_node_num;
    temp->Next = (*query_node_ptr)->Node_List;
    (*query_node_ptr)->Node_List = temp;
    insert_node_in_index(new_node_num,*query_node_ptr,temp);
    return(temp);
}
/*-----------------------------------------------------------*/


static      create_zmob_entry(zmob,zmob_ptr)

    ZMOB_MACHINE                zmob;
    ZMOB_PARTIAL_LIST_ENTRY     **zmob_ptr;
```

```
{
    ZMOB_PARTIAL_LIST_ENTRY     *temp;

#   ifdef LOWLEVEL
        fprintf(debugf,"   Entering create_zmob_entry0);
#   endif
    temp = Alloc_ZMOB_PARTIAL_LIST_ENTRY();
    temp->Zmob = zmob;
    temp->Query_List = NULL;
    temp->Next_Zmob = Head_Zmob_Partial_List.Next_Zmob;
    Head_Zmob_Partial_List.Next_Zmob = temp;
    *zmob_ptr = temp;
}
/*-------------------------------------------------------------*/

static create_query_node_entry(query,query_node_ptr,zmob_ptr)

    MESSAGE_ID                  query;
    QUERY_LIST_ENTRY            **query_node_ptr;
    ZMOB_PARTIAL_LIST_ENTRY     **zmob_ptr;

{
    QUERY_LIST_ENTRY            *temp;

#   ifdef LOWLEVEL
        fprintf(debugf,
                "    Entering create_query_node_entry0);
#   endif
    temp = Alloc_QUERY_LIST_ENTRY();
    temp->Node_List = NULL;
    temp->Query_Number = query;
    temp->Next_Query = (*zmob_ptr)->Query_List;
    (*zmob_ptr)->Query_List = temp;
    *query_node_ptr = temp;
    clear_index(temp);
}
/*-------------------------------------------------------------*/

PARTIAL_TREE_NODE   *Create_New_Partial_Tree(zmob,query,
                                             new_node_num,
                                             zmob_ptr,
                                             query_node_ptr)

    ZMOB_MACHINE                zmob;
    MESSAGE_ID                  query;
    CLAUSE_NAME                 new_node_num;
    ZMOB_PARTIAL_LIST_ENTRY     **zmob_ptr;
    QUERY_LIST_ENTRY            **query_node_ptr;

{
    PARTIAL_TREE_NODE           *tree_node;

#   ifdef PARTIALSDEBUG
```

```
        fprintf(debugf," Enter Create_New_Partial_Tree0);
#   endif
    if (*zmob_ptr == NULL) then
    {
        create_zmob_entry(zmob,zmob_ptr);
    }
    if (*query_node_ptr == NULL) then
    {
        create_query_node_entry(query,query_node_ptr,
                                                zmob_ptr);
    }
    tree_node = create_partial_tree_node(query_node_ptr,
                                                new_node_num);
    return(tree_node);
}
/*-------------------------------------------------------------*/

PARTIAL_TREE_NODE   *Create_Child_Of_Parent(parent,
                                    new_node_num,query_node)

    PARTIAL_TREE_NODE           *parent;
    CLAUSE_NAME                 new_node_num;
    QUERY_LIST_ENTRY            *query_node;

{
    PARTIAL_TREE_NODE           *temp;

#   ifdef PARTIALSDEBUG
        fprintf(debugf,"  Enter Create_Child_Of_Parent0);
#   endif
    temp = Alloc_PARTIAL_TREE_NODE();
    temp->Next = query_node->Node_List;
    query_node->Node_List = temp;
    temp->Partial = NULL;
    temp->Parent = parent; /* 10 Aug */
    temp->Node_Number = new_node_num;
    insert_node_in_index(new_node_num,query_node,temp);
    return(temp);
}
/*-------------------------------------------------------------*/

static  BOOLEAN     find_parent(parent,parent_ptr,query_node)

    CLAUSE_NAME                 parent;
    PARTIAL_TREE_NODE           **parent_ptr;
    QUERY_LIST_ENTRY            *query_node;
{
    PARTIAL_TREE_NODE           *node_list;
    BOOLEAN                     parent_found = FALSE;

    node_list = query_node->Node_Index[parent];
    if (node_list ==  NULL) then return(FALSE);
    if (node_list->Node_Number == parent) then
```

```
    {
        parent_found = TRUE;
        *parent_ptr = node_list;
#       ifdef PARTIALSDEBUG
            fprintf(debugf,
                    "Found parent %d in index0,
                                node_list->Node_Number);
#           ifdef LOWLEVEL
                Print_Partial_List(node_list->Partial);
#           endif
#       endif
    }
    else
    {
#       ifdef PARTIALSDEBUG
            fprintf(debugf,"Searching for parent node0);
#       endif
        node_list = query_node->Node_List;
        while ((!parent_found) && (node_list != NULL))
        {
            if (node_list->Node_Number == parent) then
            {
#               ifdef PARTIALSDEBUG
                    fprintf(debugf,
                                "Parent found in search0);
#               endif
                parent_found = TRUE;
                *parent_ptr = node_list;
            }
            else
            {
                node_list = node_list->Next;
            }
        }
    }
    return(parent_found);
}
/*-------------------------------------------------------*/

    BOOLEAN     Find_Query_Node(query,query_node_ptr,zmob)

    MESSAGE_ID                      query;
    QUERY_LIST_ENTRY                **query_node_ptr;
    ZMOB_PARTIAL_LIST_ENTRY         *zmob;

{
    QUERY_LIST_ENTRY                *temp;
    BOOLEAN                         query_found = FALSE;

#   ifdef PARTIALSDEBUG
        fprintf(debugf,"   Enter Find_Query_Node0);
#   endif
    temp = zmob->Query_List;
```

```
    while ((!query_found) && (temp != NULL))
    {
        if (temp->Query_Number == query) then
        {
            query_found = TRUE;
            *query_node_ptr = temp;
        }
        else
        {
            temp = temp->Next_Query;
        }
    }
    return(query_found);
}
/*-------------------------------------------------------------*/

BOOLEAN     Find_Zmob(zmob,zmob_ptr)

    ZMOB_MACHINE                    zmob;
    ZMOB_PARTIAL_LIST_ENTRY         **zmob_ptr;
{
    ZMOB_PARTIAL_LIST_ENTRY         *temp;
    BOOLEAN                         zmob_found = FALSE;

#   ifdef LOWLEVEL
        fprintf(debugf,"   Enter Find_Zmob0);
#   endif
    temp = Head_Zmob_Partial_List.Next_Zmob;
    while ((temp != NULL) && (!zmob_found))
    {
        if (temp->Zmob == zmob) then
        {
            zmob_found = TRUE;
            *zmob_ptr = temp;
        }
        else
        {
            temp = temp->Next_Zmob;
        }
    }
    return(zmob_found);
}
/*-------------------------------------------------------------*/

BOOLEAN     Find_Parent_In_Tree(zmob,query,parent,zmob_ptr,
                                query_node_ptr,
                                    parent_ptr)

    ZMOB_MACHINE                    zmob;
    MESSAGE_ID                      query;
    CLAUSE_NAME                     parent;
    ZMOB_PARTIAL_LIST_ENTRY         **zmob_ptr;
    QUERY_LIST_ENTRY                **query_node_ptr;
```

```
    PARTIAL_TREE_NODE           **parent_ptr;

{
    BOOLEAN                     found;

#   ifdef PARTIALSDEBUG
        fprintf(debugf,"   Enter Find_Parent_In_Tree0);
#   endif
    *zmob_ptr = NULL;
    *query_node_ptr = NULL;
    *parent_ptr = NULL;
    found = Find_Zmob(zmob,zmob_ptr);
    if (found) then
    {
        found = Find_Query_Node(query,query_node_ptr,
                                            *zmob_ptr);
        if (found) then
        {
            found = find_parent(parent,parent_ptr,
                                    *query_node_ptr);
        }
    }

#       ifdef PARTIALSDEBUG
            if (found) then
            {
                fprintf(debugf,
                    "  Parent node found in Find_Parent0);
            }
            else
            {
                fprintf(debugf,
                    "Parent node NOT found: Find_Parent0);
            }
#       endif
    return(found);
}
/*-------------------------------------------------------*/

Add_Partial_Constraint_To_Node(node,partial_list)

/*  add list to front of existing list */

    PARTIAL_TREE_NODE           *node;
    NODE_PARTIAL_LIST_ENTRY     *partial_list;

{

    NODE_PARTIAL_LIST_ENTRY     *temp,
                                *end;

#   ifdef PARTIALSDEBUG
        fprintf(debugf,
```

```
                "Enter Add_Partial_Constraint_To_Node0);
#   endif
    if (partial_list != NULL) then
    {
        end = partial_list;
        while (end->Next_Partial != NULL)
        {
            end = end->Next_Partial;
        }
        end->Next_Partial = node->Partial;
        node->Partial = partial_list;
#       ifdef PARTIALSDEBUG
#           ifdef LOWLEVEL
                Print_Partial_List(node->Partial);
#           endif
#       endif
    }
}
/*-------------------------------------------------------*/

Replace_Partial_By_Partial_List(old_partial,new_partial,
                                 tree_node)

    NODE_PARTIAL_LIST_ENTRY             *old_partial,
                                        *new_partial;
    PARTIAL_TREE_NODE                   *tree_node;

{
    NODE_PARTIAL_LIST_ENTRY             *end_new_partial,
                                        *predecessor;

#   ifdef PARTIALSDEBUG
        fprintf(debugf,
                "Enter Replace_Partial_By_Partial_List0);
#   endif
    end_new_partial = new_partial;
    while (end_new_partial->Next_Partial != NULL)
    {
        end_new_partial = end_new_partial->Next_Partial;
    }
    if (tree_node->Partial == old_partial) then
    {
        tree_node->Partial = new_partial;
        end_new_partial->Next_Partial =
                                old_partial->Next_Partial;
    }
    else
    {
        predecessor = tree_node->Partial;
        while (predecessor->Next_Partial != old_partial)
        {
            predecessor = predecessor->Next_Partial;
        }
```

```
        predecessor->Next_Partial = new_partial;
        end_new_partial->Next_Partial =
                            old_partial->Next_Partial;
    }
    old_partial->Next_Partial = NULL;
    Dealloc_NODE_PARTIAL_LIST_ENTRY_List(old_partial);
}
/*-------------------------------------------------------*/

Remove_Node_From_Partial_Tree(node,query)

    PARTIAL_TREE_NODE                       *node;
    QUERY_LIST_ENTRY                        *query;

{
    CLAUSE_NAME                             node_num;
    PARTIAL_TREE_NODE                       *temp;

#   ifdef PARTIALSDEBUG
        fprintf(debugf,
            "   Enter Remove_Node_From_Partial_Tree0);
#   endif
    node_num = node->Node_Number;
    query->Node_Index[node_num] = NULL;
    if (query->Node_List == node) then
    {
        query->Node_List = query->Node_List->Next;
    }
    else
    {
        temp = query->Node_List;
        while (temp->Next != node)
        {
            temp = temp->Next;
        }
        temp->Next = node->Next;
    }
    node->Next = NULL;
    Dealloc_PARTIAL_TREE_NODE(node);
}
/*-------------------------------------------------------*/

Remove_Query_From_Query_List(zmob,query)

    ZMOB_PARTIAL_LIST_ENTRY         *zmob;
    QUERY_LIST_ENTRY                *query;
{

    QUERY_LIST_ENTRY                *pred_query;

#   ifdef PARTIALSDEBUG
        fprintf(debugf,
                "Enter Remove_Query_From_Query_List0);
```

```
#   endif
    pred_query = zmob->Query_List;
    if (pred_query == query) then
    {
        zmob->Query_List = query->Next_Query;
    }
    else
    {
        while (pred_query->Next_Query != query)
        {
            pred_query = pred_query->Next_Query;
        }
        pred_query->Next_Query = query->Next_Query;
    }
    query->Next_Query = NULL;
}
/*-------------------------------------------------------*/

Remove_Zmob_From_Zmob_Partial_List(zmob)

    /* uses global Headnode */

    ZMOB_PARTIAL_LIST_ENTRY         *zmob;
{

    ZMOB_PARTIAL_LIST_ENTRY         *pred_zmob;

#   ifdef PARTIALSDEBUG
        fprintf(debugf,
        "Enter Remove_Zmob_From_Partial_List entry0);
#   endif
    pred_zmob = Head_Zmob_Partial_List.Next_Zmob;
    if (pred_zmob == zmob) then
    {
        Head_Zmob_Partial_List.Next_Zmob = zmob->Next_Zmob;
    }
    else
    {
        while (pred_zmob->Next_Zmob != zmob)
        {
            pred_zmob = pred_zmob->Next_Zmob;
        }
        pred_zmob->Next_Zmob = zmob->Next_Zmob;
    }
    zmob->Next_Zmob = NULL;
}
/*-------------------------------------------------------*/

Erase_Zmob_Query_List_Entry(zmob,query)

    /* only called when zmob and query exist */

    ZMOB_PARTIAL_LIST_ENTRY         *zmob;
```

```
    QUERY_LIST_ENTRY                    *query;

{
#   ifdef PARTIALSDEBUG
        fprintf(debugf,
                "Enter Erase_Zmob_Query_List_Entry0);
#   endif
    Remove_Query_From_Query_List(zmob,query);
    Dealloc_QUERY_LIST_ENTRY(query);
    if (zmob->Query_List == NULL) then
    {
#       ifdef PARTIALSDEBUG
            fprintf(debugf,
                    "Zmob's query list is empty0);
            fprintf(debugf,
                    "So removing it from Zmob list0);
#       endif
        Remove_Zmob_From_Zmob_Partial_List(zmob);
        Dealloc_ZMOB_PARTIAL_LIST_ENTRY(zmob);
    }
}
/*-------------------------------------------------------------*/
```

```
/* File predindex.c: Contains all routines to manipulate
   the predicate index.                                    */

#ifdef CMDEBUG
#   define PREDINDEXDEBUG
#endif

#include "cm.h"

extern  FILE                            *debugf;
extern  PREDICATE_ENTRY                 *Predicate_Table;
extern  CONSTRAINT_ENTRY                *Constraint_Table;
extern  int                             Number_Predicates;
extern  CONTAINING_CONSTRAINTS
                            *Alloc_CONTAINING_CONSTRAINTS();

/*---------------------------------------------------------*/

CONTAINING_CONSTRAINTS  *Merge_Containing_Constraints(list1,
                                                      list2)

/* add list two to the end of list one */

    CONTAINING_CONSTRAINTS          *list1,
                                    *list2;

{
    CONTAINING_CONSTRAINTS          *temp,
                                    *end;

    if (list1 == NULL) then
    {
        return(list2);
    }
    else
    {
        temp = list1;
        end = NULL;
        while (temp != NULL)
        {
            end = temp;
            temp = temp->Next_Constraint;
        }
        end->Next_Constraint = list2;
        return(list1);
    }
}
/*---------------------------------------------------------*/

static CONTAINING_CONSTRAINTS *Copy_Containing_Constraints(
                                                      list)

    CONTAINING_CONSTRAINTS          *list;
```

```
{
    CONTAINING_CONSTRAINTS          *temp,
                                    headnode;

    temp = &headnode;
    while (list != NULL)
    {
        temp->Next_Constraint =
                            Alloc_CONTAINING_CONSTRAINTS();
        temp = temp->Next_Constraint;
        temp->Next_Constraint = NULL;
        temp->Constraint_Id = list->Constraint_Id;
        list = list->Next_Constraint;
    }
    return(headnode.Next_Constraint);
}
/*-------------------------------------------------------*/

static  CONTAINING_CONSTRAINTS  *search_predicate_table(name)

    LITERAL_NAME                    name;

{
    int                             i;

    for (i = 0; i < Number_Predicates; i++)
    {
        if (Predicate_Table[i].Name == name) then
        {
            return(Predicate_Table[i].Constraint_List);
        }
    }
    return(NULL);
}
/*-------------------------------------------------------*/

CONTAINING_CONSTRAINTS  *Find_All_Constraints_To_Check(body)

    LITERAL                         *body;

{
    CONTAINING_CONSTRAINTS          answer,
                                    *list;


#   ifdef PREDINDEXDEBUG
        fprintf(debugf,
                "Enter Find_All_Constraints_To Check0);
#   endif
    answer.Next_Constraint = NULL;
    while (body != NULL)
    {
        list = search_predicate_table(body->Name);
```

```
        if (list != NULL) then
        {
        list = Copy_Containing_Constraints(list);
        answer.Next_Constraint =
            Merge_Containing_Constraints(
                                    answer.Next_Constraint,
                                                list);
        }
        body = body->Next_Literal;
    }

#   ifdef PREDINDEXDEBUG
        if (answer.Next_Constraint == NULL) then
        {
            fprintf(debugf,
                    "  NO constraints found to check0);
        }
        else
        {
            list = answer.Next_Constraint;
            while (list != NULL)
            {
                fprintf(debugf,"Constraint to check = %d0,
                                    list->Constraint_Id);
                list = list->Next_Constraint;
            }
        }
#   endif
    return(answer.Next_Constraint);
}
/*-------------------------------------------------------*/
```

```
/* File startup.c: Contains the command line options
   handler. This routine is common to most PRISM programs
   and was copied and modified for the Constraint Machine */

#ifdef CMDEBUG
#   define  STARTUPDEBUG
#   define  VAXDEBUG
#endif

# include   "cm.h"

extern  FILE            *debugf;
extern  ZMOB_MACHINE    Myself;
extern  BOOLEAN         Using_Belt;
extern  BOOLEAN         Occur_Check;
extern  FILE            *Input_File;
extern  FILE            *Output_File;
extern  PORT_ID         Receive_Port;
extern  PORT_ID         Transmit_Port;
extern  MESSAGE         *Alloc_MESSAGE();
extern  ZMOB_MESSAGE    *Make_Output_Message();

#ifdef VAXDEBUG
    extern TAG_TABLE    Tag_File_Table[];
    extern              Dump_Constraint_Table();
    extern              Dump_Predicate_Table();
#endif

/*-------------------------------------------------------------*/

static get_options(argc,argv)
  int    argc;
  char  *argv[];
{
  int           arg_num = 0;
  FILE          *cm_file;
  BOOLEAN       saw_F = FALSE;
  BOOLEAN       saw_N = FALSE;
#  ifdef VAXDEBUG
       FILE     *tag_file = NULL;
#  endif

#  ifdef STARTUPDEBUG
     fprintf(debugf,"CM Getting Options0);
#  endif
  while (++arg_num < argc)
  {
    if (argv[arg_num][0] == '-') then
    {
      switch (argv[arg_num][1])
      {
/*        case 'b':
          Using_Belt = TRUE;
```

```
        Receive_Port =
                Allocate_Input_Port(argv[++arg_num]);
        Transmit_Port =
                Allocate_Output_Port(argv[++arg_num]);
        break;                          removed due to 4.2  */
      case 'd':
        debugf = fopen(argv[++arg_num],"w");
        if (debugf == NULL) then
        {
          debugf = stderr;
          fprintf(debugf,
          "Could not open %s,assumed terminal debug0,
                argv[arg_num]);
        }
        break;
      case 'f':
        cm_file = fopen(argv[++arg_num],"r");
        if (cm_file == NULL) then
        {
          fprintf(debugf,
            "Could not open %s0,argv[arg_num]);
          Abort(FALSE);
        }
        saw_F = TRUE;
        break;
      case 'i':
        Input_File = fopen(argv[++arg_num],"r");
        if (Input_File == NULL) then
        {
          fprintf(debugf,
            "Cannot open file %s0,argv[arg_num]);
          Abort(FALSE);
        }
#        ifdef STARTUPDEBUG
        else
        {
          fprintf(debugf,
            "Opened input file: %s0,argv[arg_num]);
        }
#        endif
        break;
      case 'n':
        Myself = atoi(argv[++arg_num]);
        saw_N = TRUE;
        break;
      case 'o':
        Output_File = fopen(argv[++arg_num],"w");
        if (Output_File == NULL) then
        {
          fprintf(debugf,
            "Cannot open file %s0,argv[arg_num]);
          Abort(FALSE);
        }
```

```
            break;
#           ifdef VAXDEBUG
          case 't':
            tag_file = fopen(argv[++arg_num],"r");
            if (tag_file == NULL) then
            {
              fprintf(debugf,
                "Could not open %s0,argv[arg_num]);
              Abort(FALSE);
            }
            fprintf(debugf,
              "Starting to read tag file0);
            Read_Tag_File(tag_file,Tag_File_Table);
            fclose(tag_file);
            break;
#           endif
          default:
            fprintf(debugf,
                "Unknown Option %c0,argv[arg_num][1]);
            break;
        }
      }
      else
      {
        fprintf(debugf,
            "Unknown Argument %s0,argv[arg_num]);
      }
    }
    if (!saw_N || !saw_F) then
    {
      fprintf(debugf,
          "CM must have an Id AND a Constraint File0);
      fprintf(debugf,
          "Usage is: cm -f d1 -n id [-d f3] [-i f4] [-o f5] ");
      fprintf(debugf,"[-b p1 p2]0);
      Abort(FALSE);
    }
    if (saw_F)
    {
      Read_Constraint_File(cm_file);
      fclose(cm_file);
    }
#   ifdef STARTUPDEBUG
      fprintf(debugf,"Options processed0);
#   endif
}
/*-------------------------------------------------------------*/

CM_Startup(argc,argv)
    int     argc;
    char    *argv[];
{
    Init_Debug();
```

```
    Init_Heap();
    get_options(argc,argv);
/*  if (Using_Belt) then
    {
        Attach_Interrupts();
    }                              removed due to 4.2 */
#   ifdef STARTUPDEBUG
        fprintf(debugf,"Start Up Initialization Done0);
#   endif
#   ifdef VAXDEBUG
        Dump_Predicate_Table();
        Dump_Constraint_Table();
#   endif
}
/*-------------------------------------------------------*/
```

```
/* File subsump.c: Contains the subsumption algorithm */

#ifdef CMDEBUG
#   define SUBSUMPDEBUG
#   undef  LOWLEVEL
#endif

# include "cm.h"

#ifdef  SUBSUMPDEBUG
    extern                          Print_Partial_List();
#endif
extern  FILE                        *debugf;
extern  SUBSTITUTION                *Unifier();
extern  LITERAL                     *Copy_Literal_List();
extern  LITERAL
                    *Copy_Literal_List_And_Remove_Literal();
extern  LITERAL                     *Remove_Literal_From_List();
extern  TERM                        *Copy_Term_List();
extern  NODE_PARTIAL_LIST_ENTRY
                            *Alloc_NODE_PARTIAL_LIST_ENTRY();
extern  SUBSTITUTION                *Alloc_SUBSTITUTION_NODE();
extern  SUBSUMPTION_NODE            *Alloc_SUBSUMPTION_NODE();
static  SUBSUMPTION_NODE            *queue[MAX_QUEUE_SIZE];
static  int                         Front = 0;
static  int                         Back = 0;
extern  SUBSUMPTION_PARTIAL_LIST
                        *Alloc_SUBSUMPTION_PARTIAL_LIST();
extern                              Dealloc_Subsumption_Tree();
extern
                    Dealloc_SUBSUMPTION_PARTIAL_LIST_List();
extern  LITERAL                     *Evaluate_One_Literal_List();

/*-------------------------------------------------------*/

static  BOOLEAN     add_queue(node)
    SUBSUMPTION_NODE                *node;
{
    int                             temp;

#   ifdef LOWLEVEL
        fprintf(debugf,"  Entering add_queue0);
#   endif
    if ((temp = ((Back + 1) % MAX_QUEUE_SIZE)) == Front)
    {
        fprintf(stderr,"Queue overflow on add 0);
        return(TRUE);
    }
    else
    {
        queue[Back] = node;
        Back = temp;
        return(FALSE);
```

```
    }
}
/*-------------------------------------------------------*/

static  BOOLEAN     empty_queue()

{
    if (Back == Front)
        return(TRUE);
    else
        return(FALSE);
}

static  SUBSUMPTION_NODE    *remove_queue()

{
    SUBSUMPTION_NODE            *temp;

#   ifdef LOWLEVEL
        fprintf(debugf,"    Entering remove_queue0);
#   endif
    if (empty_queue()) then
    {
        fprintf(stderr,
                "Attempted to remove from empty queue0);
        return(NULL);
    }
    else
    {
        temp = queue[Front];
        Front = (Front + 1) % MAX_QUEUE_SIZE;
        return(temp);
    }
}
/*-------------------------------------------------------*/

NODE_PARTIAL_LIST_ENTRY     *Subsumption(body,constraint,
                                         subsumption_answer)

/* Input two literal lists. Output all partial subsumptions
   with all evaluable EQ predicates evaluated.            */

    LITERAL                 *body;
    LITERAL                 *constraint;
    SUBSUMPTION_ANSWER      *subsumption_answer;

{

    SUBSUMPTION_NODE        root;
    SUBSUMPTION_NODE        *parent, *child, *last_child;
    SUBSTITUTION            *mgu_answer;
    BOOLEAN                 unifiable;
    BOOLEAN                 done = FALSE;
```

```
    BOOLEAN                 subsumption_nodes_created = FALSE;
    TERM                    *constraint_term_list,
                            *body_term_list;
    LITERAL                 *temp, *i, *j,
                            *partial_constraint;
    BOOLEAN                 error_flag;
    SUBSUMPTION_STATUS      status, eval_status;
    SUBSUMPTION_PARTIAL_LIST    answer_node_list,
                                *next_anwer_node;
    NODE_PARTIAL_LIST_ENTRY     partial_list, *next_partial;

#   ifdef SUBSUMPDEBUG
        fprintf(debugf,"Entering Subsumption algorithm0);
#   endif
    Front = Back = 0;
    partial_list.Next_Partial = NULL;
    root.Constraint = constraint;
    root.Body = body;
    root.Child = root.Sib = NULL;
    next_anwer_node = &answer_node_list;
    next_partial = &partial_list;
    error_flag = add_queue(&root);
    while (!empty_queue())
    {
        error_flag = ((parent = remove_queue()) == NULL);
        if (error_flag) fprintf(debugf,
                        "error_flag set in Subsumption0);
        status = CREATING_CHILD;
        for (i = parent->Constraint; i != NULL;
                                        i = i->Next_Literal)
        {
            for (j = parent->Body; j != NULL;
                                    j = j->Next_Literal)
            {
#               ifdef LOWLEVEL
                    fprintf(debugf,
                        " Body literal name = %d,",j->Name);
                    fprintf(debugf,
                            "Constraint literal name= %d0,
                                                i->Name);
#               endif
                if (i->Name == j->Name) then
                {
#                   ifdef LOWLEVEL
                        fprintf(debugf,
                                    "Predicates match0);
#                   endif
                    constraint_term_list =
                            Copy_Term_List(i->Arg_List);
                    body_term_list =
                                Copy_Term_List(j->Arg_List);
                    mgu_answer=Unifier(constraint_term_list,
                                        body_term_list,
```

```
                                                &unifiable);
                        if (unifiable) then
                        {
                            subsumption_nodes_created = TRUE;
                            child = Alloc_SUBSUMPTION_NODE();
                            error_flag = add_queue(child);
                            if (error_flag)
                                fprintf(stderr,
                                "Full queue add in Subsump0);
                    temp = Copy_Literal_List_And_Remove_Literal(
                                    parent->Constraint,i);
                            Apply_Sub_List(temp,mgu_answer);
                            child->Constraint = temp;
                    temp = Copy_Literal_List_And_Remove_Literal(
                                                parent->Body,j);
                            child->Body = temp;
                            child->Sib = child->Child = NULL;
                            if (status == CREATING_CHILD) then
                            {
                                parent->Child = child;
                                last_child = child;
                            }
                            else
                            {
                                last_child->Sib = child;
                                last_child = child;
                            }
                            status = CREATING_SIB;
                        }
                    }
                    else
                    {
#                       ifdef LOWLEVEL
                            fprintf(debugf,
                                "Predicates do not match0);
#                       endif
                    }
                }
            } /* end for */
            if (status == CREATING_CHILD) then
            {
#               ifdef LOWLEVEL
                    fprintf(debugf,
                        "Adding parent node to answer list0);
#               endif
                next_anwer_node->Next =
                                Alloc_SUBSUMPTION_PARTIAL_LIST();
                next_anwer_node = next_anwer_node->Next;
                next_anwer_node->Next = NULL;
                next_anwer_node->Answer = parent;
            }
        } /* end while */
```

```
    /* now answer_node_list has pointers to all leaf nodes
       to return values                                        */

    if ((answer_node_list.Next)->Answer == &root) then
    {
#       ifdef SUBSUMPDEBUG
            fprintf(debugf,
                    "NO partial subsumption occurred0);
#       endif
        *subsumption_answer = OKAY;
    }
    else
    {
        next_anwer_node = answer_node_list.Next;
        *subsumption_answer = PARTIAL;
#       ifdef SUBSUMPDEBUG
            fprintf(debugf,
                    "A PARTIAL Subsumption occurred....0);
#       endif
        while ((next_anwer_node != NULL) && (! done))
        {
            partial_constraint =
                        next_anwer_node->Answer->Constraint;
            next_anwer_node->Answer->Constraint = NULL;
            if (partial_constraint == NULL) then
            {       /* empty constraint = full subsumption */

#               ifdef SUBSUMPDEBUG
                    fprintf(debugf,
                        "FULL subsumption has occurred0);
#               endif
                *subsumption_answer = FULL;
                done = TRUE;
            }
            else /* add constraint to list */
            {
                partial_constraint =
                    Evaluate_One_Literal_List(
                                        partial_constraint,
                                            &eval_status);
                switch (eval_status)
                {
                    case FULL:
                        *subsumption_answer = FULL;
                        done = TRUE;
                        break;
                    case OKAY:
                        next_partial->Next_Partial =
                            Alloc_NODE_PARTIAL_LIST_ENTRY();
                        next_partial =
                                next_partial->Next_Partial;
                        next_partial->Next_Partial = NULL;
                        next_partial->Constraint =
```

```
                                        partial_constraint;
                        next_anwer_node =
                                        next_anwer_node->Next;
                        break;
                    case NO_VIOLATION_POSSIBLE:
                        next_anwer_node =
                                        next_anwer_node->Next;
                        break;
                    case PARTIAL:
                        fprintf(debugf,
                        "PARTIAL ret by Eval One Lit0);
                        break;
                    default:
                        fprintf(debugf,
                        "Unknown eval_status in Subsump0);
                }
            }
        } /* end while */
        if ((partial_list.Next_Partial == NULL) &&
                        (*subsumption_answer != FULL))
            *subsumption_answer = OKAY;
    }
    if (subsumption_nodes_created) then
    {
    Dealloc_Subsumption_Tree(root.Child);
    Dealloc_SUBSUMPTION_PARTIAL_LIST_List(
                                        answer_node_list.Next);
    }
#   ifdef SUBSUMPDEBUG
        fprintf(debugf,"Exiting Subsumption algorithm0);
#       ifdef LOWLEVEL
            fprintf(debugf,"  Printing partial list....0);
            Print_Partial_List(partial_list.Next_Partial);
#       endif
#   endif
    return(partial_list.Next_Partial);
}
/*-------------------------------------------------------*/
```

BIBLIOGRAPHY

[Chang 1973]

Chang,C.L., Lee,R.C.T., Symbolic Logic and Mechanical Theorem Proving, Academic Press, New York, 1973.

[Clocksin 1981]

Clocksin,W.F., Mellish,C.S., Programming in Prolog, Springer-Verlag, New York, N.Y., 1981.

[Eisinger 1981]

Eisinger,N., Kasif,S., Minker,J., "Logic Programming: A Parallel Approach", TR-1124, Dept. of Computer Science, University of Maryland, December 1981, College Park, Maryland.

[Futo 1984]

Futo,I., "A Constraint Machine to Control Parallel Search of PRISM", Institute for Coordination of Computer Techniques, H-1368 POB 224 Hungary, March 1984, (in preparation).

[Kasif 1983]

Kasif,S., Kohli,M., Minker,J., "A Parallel Inference System for Problem Solving", TR-1243 Dept. of Computer Science, University of Maryland, January 1983, College Park, Maryland.

[Kohli 1983]

Kohli,M., Minker,J., "Integrity Constraints to Control

Execution of Logic Programs", 2nd International Workshop on Logic Programming, Portugal, 1983.

[Kowalski 1974]

Kowalski, R.A., "Predicate Logic as Programming Language", Proceedings of IFIP 74, pp 569-574, North-Holland, Amsterdam, 1974.

[Kowalski 1979]

Kowalski, R.A., Logic for Problem Solving, Elsevier North Holland Inc., 1979, New York.

[Johnson 1978]

Joshnson,S.C., "Yacc: Yet Another Compiler Compiler", Bell Laboratories, Murray Hill, New Jersey, 1978.

[Minker 1982]

Minker,J., Asper,C., Dehe,C., et al, "Functional Specification of the ZMOB Parallel Problem Solving System", TN Z-1, Dept. of Computer Science, University of Maryland, August 1982, (in preparation) College Park, Maryland.

[Pereira 1978]

Pereira, L.M., Monteiro, L.F., "The Semantics of Parallelism and Co-Routining in Logic Programming", Divisao de Informatica, Laboratorio Nacional de Egenhario Civil, December 1978, Lisbon.

[Rieger 1980]

Rieger, C., Bane, J., Trigg, R., "ZMOB : A Highly Parallel Multiprocessor", TR-911, Dept. of Computer Science, University of Maryland, May 1980, College Park, Maryland.

[Rieger 1981a]

Rieger, C., Trigg, R., Bane, J., "ZMOB : A New Computing Engine for AI", TR-1028, Dept. of Computer Science, University of Maryland, March 1981, College Park, Maryland.

[Rieger 1981b]

Rieger, C., "ZMOB : Hardware from a User's Point of View", TR-1042, Dept. of Computer Science, University of Maryland, April 1981, College Park, Maryland.

[Roberts 1977]

Roberts,G.M., "An Implementation of PROLOG", M.S. Thesis, University of Waterloo, 1977.

[Robinson 1965]

Robinson,J.A., "A Machine Oriented Logic Based on the Resolution Principle", J. ACM 12, January 1965.

[Roussel 1975]

Rousell,P., PROLOG: Manuel De Reference et d'Utilization, Groupe d'Intelligence Artificielle, University d'Aix-Marseille, Luminy, 1975.

[van Emden 1976]

van Emden, M.H., Lucena, G.H., de Silva, H.M., "Predicate Logic as a Language for Parallel Programming", Research Report, Dept. of Computer Science, Univ. of Waterloo, Ontario.